CS/ECE 752: Advanced Computer Architecture I

Source: XKCD

Professor Matthew D. Sinclair

Advanced OOO

Slide History/Attribution Diagram:

Today

- Last Time: Two more OOO Schemes:
  - P6: ARF stores committed state, ROB/RS speculative State
  - R10K: All State in merge register file
- Realistic Memory Dependences
  - LSQ Implementation: Conservative & Intelligent
- Taste of advanced OOO techniques (**likely will not be covered – extra material for those who are interested**)
  - Hide Long-latency Memory Access: Continual Flow Pipelines
  - Reduce Critical Scheduling Loop: Speculative Scheduling
  - Break dataflow limit with value prediction

OOO Performance Considerations

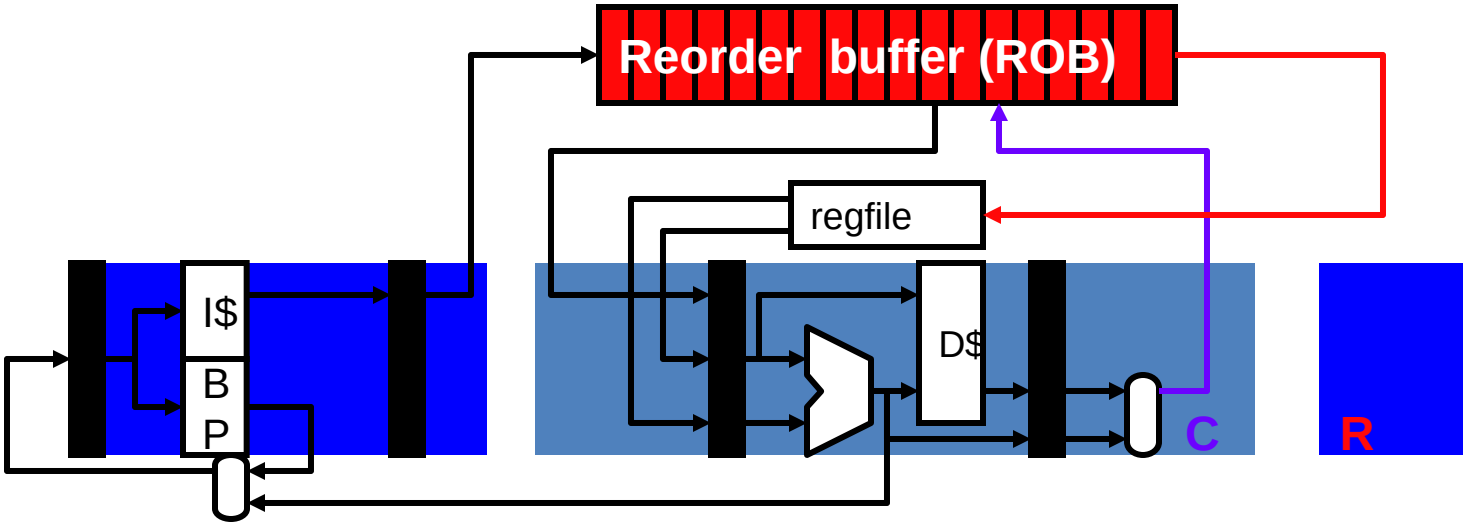
- Issue Width (Maximum Performance)
  - Governed by: "Superscalar Width" of Fetch/Decode/Rename/Dispatch/ Issue/**LSQ**/Reg-File/Wakeup/Commit
- Instruction Window (Scheduling Scope)
  - Governed by: Size of Reservation Stations ("Instruction Queue"), Reorder Buffer (ROB), Register Files, **LSQ**
  - Branch prediction capability
- Dependence Latency (Critical Path)
  - Governed by:
    - Data dependences: Functional Unit Latencies
    - Micro-arch Dependences: Length of critical loops (Fetch Loop, Commit Loop, Scheduler Loop, Branch Resolution Loop)

Where do we put Speculation? (control, memory dependence?)

Where do we put vector-instructions?

Where do we put Caching?

Complete and Retire (Review)



- **Complete (C)**: first part of writeback (W)
  - Completed insns write results into ROB
  - \+ Out-of-order: **wait** doesn't back-propagate to younger insns
- **Retire (R)**: aka commit, graduate
  - ROB writes results to register file
  - In order: **stall** back-propagates to younger insns

In-Class Assignment

- With a partner, answer the following questions:
  - What makes it so difficult to get around the techniques proposed in the Moshovos paper?

- In 3 minutes we'll discuss as a class

Blind appr. ok? ⇒ cut deeper/wider now
No good, high pert. alts.

Out of Order Memory Operations

- All insns are easy in out-of-order…
  - Register inputs only
  - Register renaming captures all dependences
  - Tags tell you exactly when you can execute
- … except loads
  - Register and memory inputs (older stores)
  - Register renaming does not tell you all dependences
  - How do loads find older in-flight stores to same address (if any)?

Load/Store Queue (LSQ)

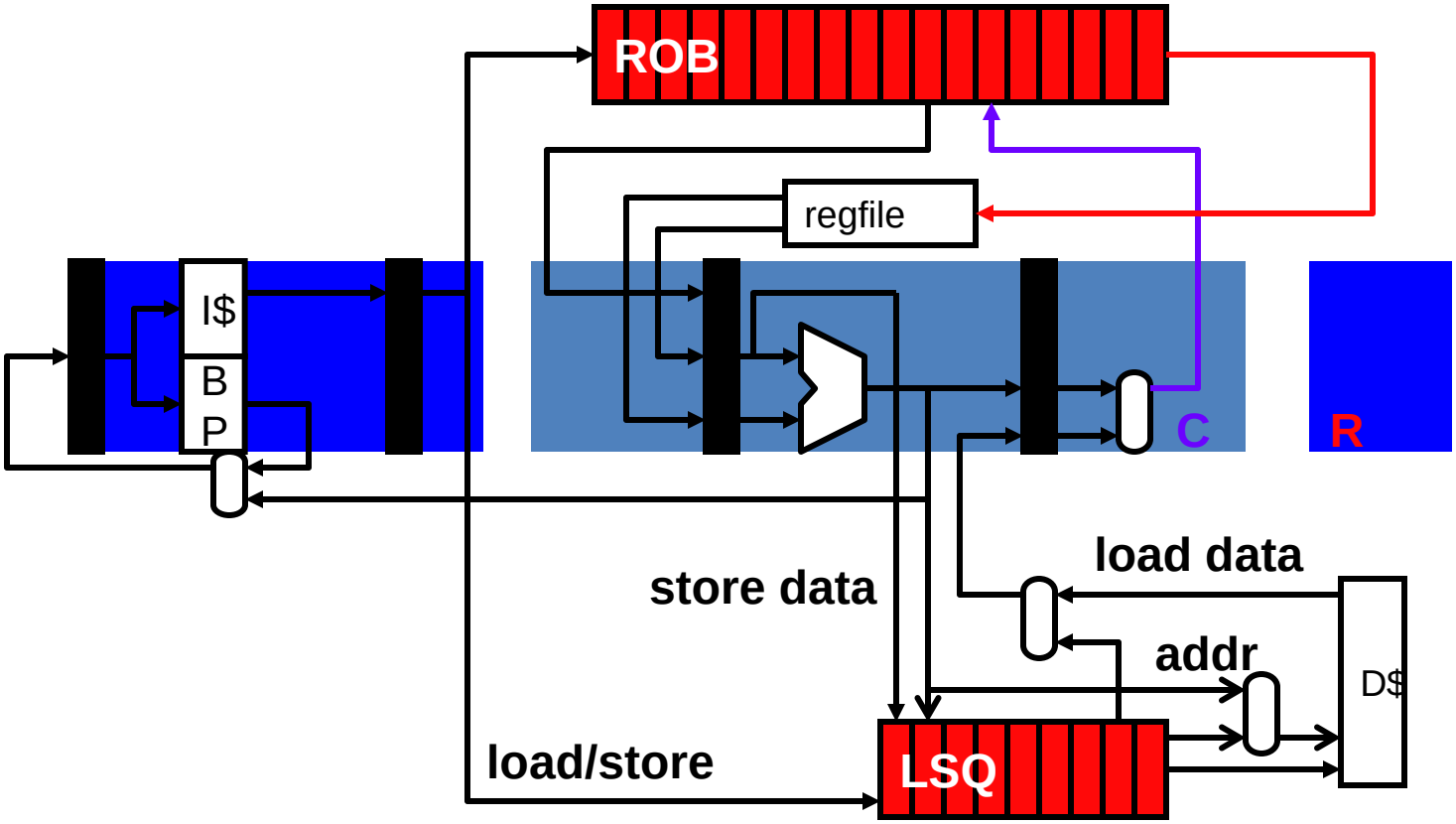
- ROB makes register writes in-order, but what about stores?

- Could we do what in-order-core does…
  … i.e., store to D$ in X stage?
  - Not even close, imprecise memory worse than imprecise registers

- **Load/store queue (LSQ)**
  - Completed stores write to LSQ
  - When store retires, head of LSQ written to D$
  - When loads execute, access LSQ and D$ in parallel
    - Forward from LSQ if older store with matching address
  - More modern design: loads and stores in separate queues
  - More on this later

head of ROB

ROB + LSQ

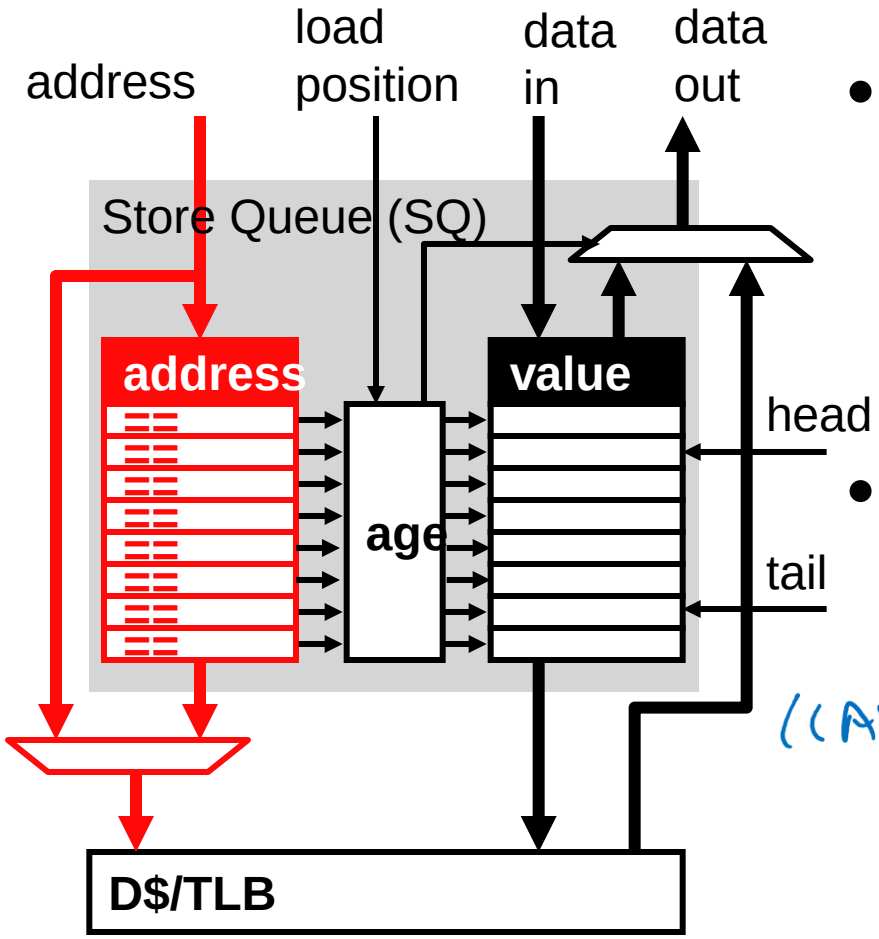


- Modulo gross simplifications, this picture is almost realistic!

Data Memory Functional Unit

- D$/TLB + structures to handle in-flight loads/stores
  - Performs four functions
  - **In-order store retirement**
    - Writes stores to D$ in order
    - Basic, implemented by store queue (SQ)
  - **Store-load forwarding**
    - Allows loads to read values from older un-retired stores
    - Also basic, also implemented by store queue (SQ)
  - **Memory ordering violation detection**
    - Checks load speculation
    - Advanced, implemented by load queue (LQ)
  - **Memory ordering violation avoidance**
    - Advanced, implemented by dependence predictors

Simple Data Memory FU: D$/TLB + SQ

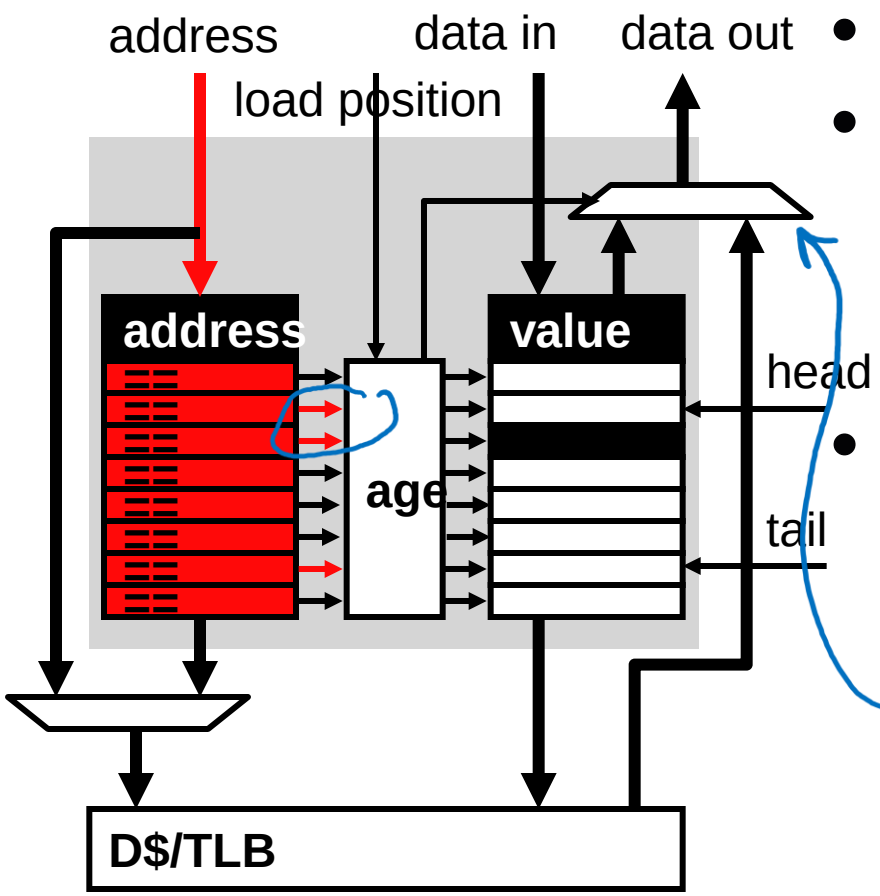


- Just like any other FU
  - 2 register inputs (addr, data in)
  - 1 register output (data out)
  - 1 non-register input (load pos)?
    - ↳ position in queue
- **Store queue (SQ)**
  - In-flight store address/value
  - In program order (like ROB)
  - Addresses associatively searchable
  - Size heuristic: 15-20% of ROB
- But what does it do?

Data Memory FU "Pipeline"

- Stores
  - **Dispatch (D)**
    - Allocate entry at SQ tail
  - **Execute (X)**
    - Write address and data into corresponding SQ slot
  - **Retire (R)**
    - Write address/data from SQ head to D$, free SQ head
- Loads
  - **Dispatch (D)**
    - Record current SQ tail as "load position"
  - **Execute (X)**
    - Where the good stuff happens

"Out-of-Order" Load Execution



- In parallel with D$ access
- **Send address to SQ**
  - Compare with all store addresses
  - CAM: like FA$, or RS tag match
  - Select all matching addresses
- **Age logic selects youngest store that is older than load**
  - Uses load position input
  - Any? load **"forwards"** value from SQ
  - None? Load gets value from D$
  - Address not known? Wait?

Need most recent prior ST for in order semantics

Conservative Load Scheduling

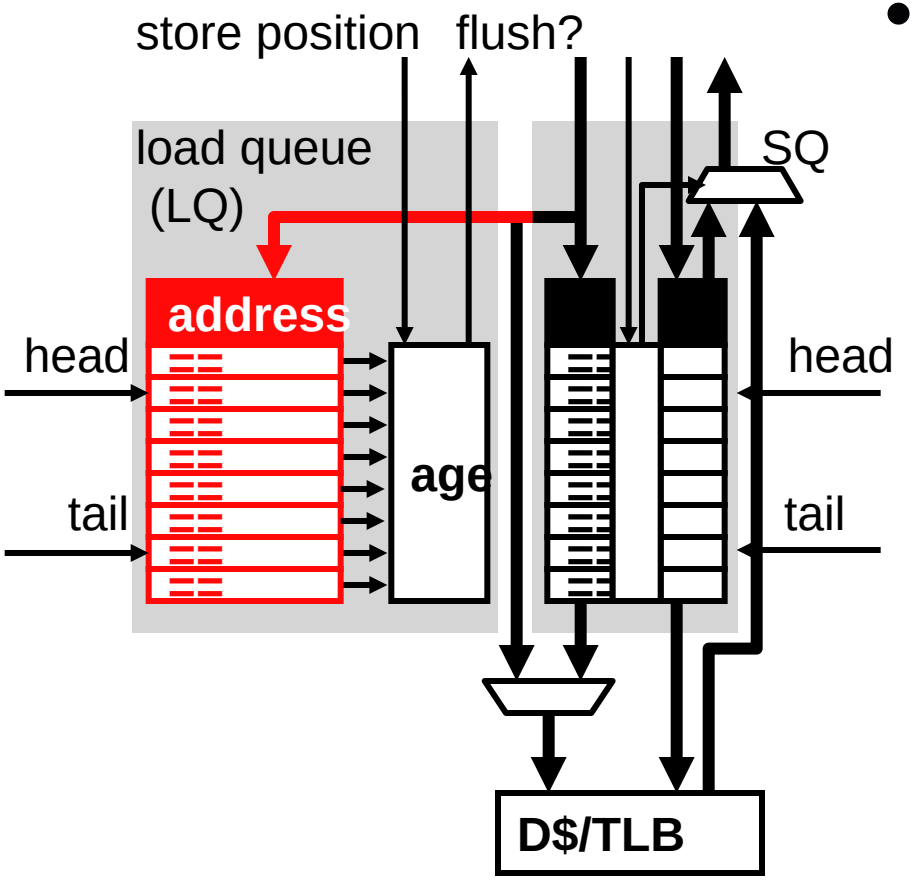
- Why "" in "out-of-order"?
  - + Load can execute out-of-order with respect to (wrt) other loads
  - + Stores can execute out-of-order wrt other stores
  - – **Loads must execute in-order wrt older stores**
    - Load execution requires knowledge of all older store addresses
    - + Simple
    - – Restricts performance
  - Used in Original P6 Implementation

Opportunistic Memory Scheduling

- Observe: on average, < 10% of loads forward from SQ
  - Even if older store address is unknown, chances are it won't match
  - Let loads execute in presence of older **"ambiguous stores"**
  + Increases performance
  - But what if ambiguous store *does* match?

- **Memory ordering violation**: load executed too early
  - Must detect…
  - And fix (e.g., by flushing/refetching insns starting at load)

D$/TLB + SQ + LQ



- **Load queue (LQ)**
  - In-flight load addresses
  - In program-order (like ROB,SQ)
  - Associatively searchable ((AM))
  - Size heuristic: 20-30% of ROB

Advanced Memory "Pipeline" (LQ Only)

- Loads
  - **Dispatch (D)**
    - Allocate entry at LQ tail
  - **Execute (X)**
    - Write address into corresponding LQ slot
- Stores
  - **Dispatch (D)**
    - Record current LQ tail as "store position"
  - **Execute (X)**
    - Where the good stuff happens

Intelligent Load Scheduling

"Futloads" [MICRO'13] ↑ '20?

- Opportunistic scheduling better than conservative…
  - + Avoids many unnecessary delays
- …but not significantly
  - – Introduces a few **flushes,** but **each is much costlier than a delay**

- Observe: loads/stores that cause violations are "stable"
  - Dependences are mostly program based, program doesn't change
  - Scheduler is deterministic

Guri Sohi

- Exploit: **intelligent load scheduling**
  - Hybridize conservative and opportunistic
  - Predict which loads, or load/store pairs will cause violations
  - Use conservative scheduling for those, opportunistic for the rest

Memory Dependence Prediction

- Store-blind prediction
  - Predict load only, wait for all older stores to execute
  - ± Simple, but a little too heavy handed
  - Example: Alpha 21264

- Store-load pair prediction
  - Predict load/store pair, wait only for one store to execute
  - ± More complex, but minimizes delay
  - Example: Store-Sets
    - Load identifies the right dynamic store in two steps
    - (1) Store-Set Table: load-PC → store-PC
    - (2) Last Store Table: store-PC → SQ index of most recent instance

Pentium III vs. Pentium4 (Processors)

Feature	Pentium III (1999)	Pentium 4 (2000)
Peak clock	800 MHz	3.4 GHz
Pipeline stages	15	22
Branch prediction	512 local + 512 BTB	2K hybrid + 2K BTB
Primary caches	16KB 4-way	8KB 4-way + 64KB T$
L2	512KB-2MB	256KB-2MB
Fetch width	16 bytes	3 μops (16 bytes on miss)
Rename/retire width	3 μops	3 μops
Execute width	5 μops	7 μops (X2)
Register renaming	P6	R10K
ROB/RS size	40/20	128/60
Load scheduling	Conservative	Intelligent
Anything else?	No	Hyperthreading

Pentium4 vs Coffee Lake (Processors)

Feature	Pentium 4 (2000)	Coffee Lake (2018)
Peak clock	3.4 GHz	5.00 GHz (Turbo)
Pipeline stages	22	14-19
Branch prediction	2K hybrid + 2K BTB	???
Primary caches	8KB 4-way + 64KB T$	32 KB, 8-way + uOp Cache
L2	256KB-2MB	256KB (up 2MB l3 per core)
Fetch width	3 μops (16 bytes on miss)	6 μops (16 byte on miss)
Rename/retire width	3 μops	6 μops (guess)
Execute width	7 μops	8 μops
Register renaming	R10K	R10K
ROB/RS size	128/60	224/97
Load scheduling	Intelligent	Intelligent
Anything else?	Hyperthreading	Multicore (Spectre/Meltdown/ Zombieload :)

Today

- Last Time: Two more OOO Schemes:
  - P6: ARF stores committed state, ROB/RS speculative State
  - R10K: All State in merge register file
- Realistic Memory Dependences
  - LSQ Implementation: Conservative & Intelligent
- Taste of advanced OOO techniques
  - Continual Flow Pipelines
  - **Speculative Scheduling**
  - Value Prediction

Architecture "Loops" Affect Critical Path

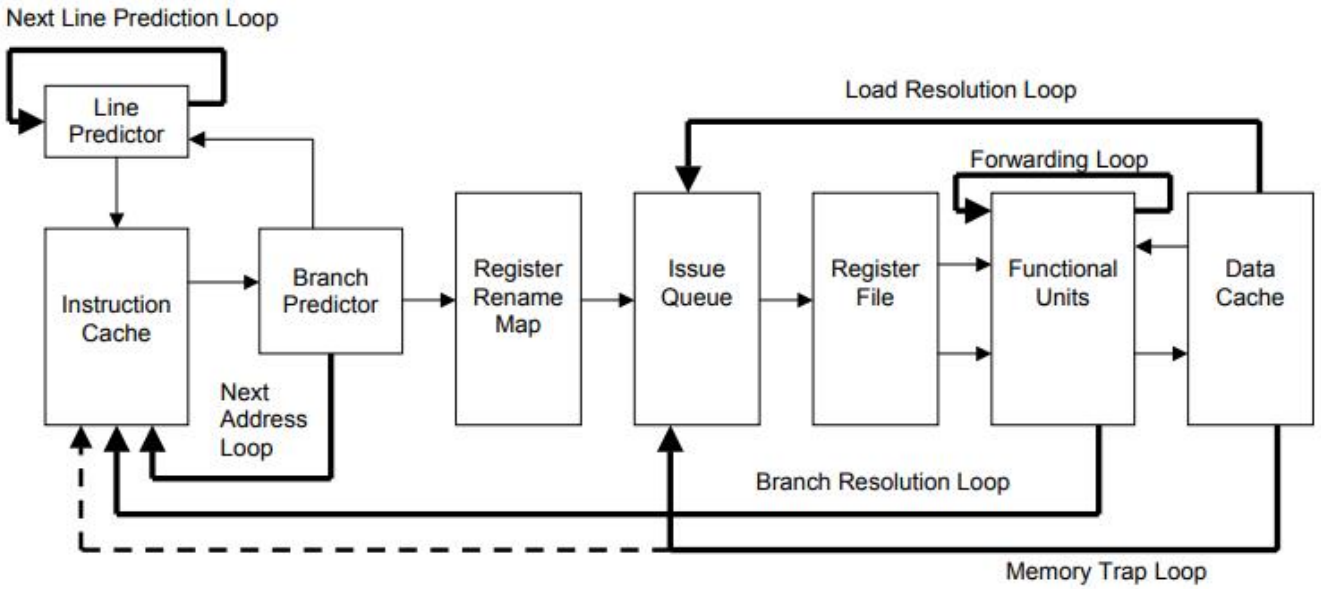


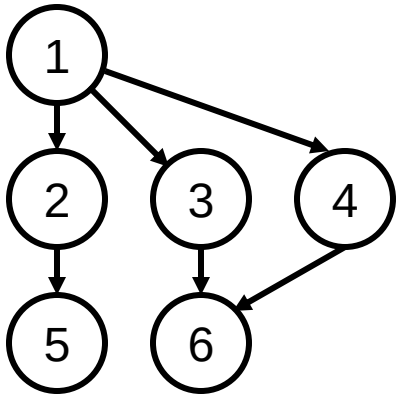
Figure 2. Examples of micro-architectural loops in the Alpha 21264 processor [3], represented by solid lines. If the recovery stage is different from the initiation stage, it is shown by a dotted line.

Loose Loops Sink Chips [HPCA 02, Borch]

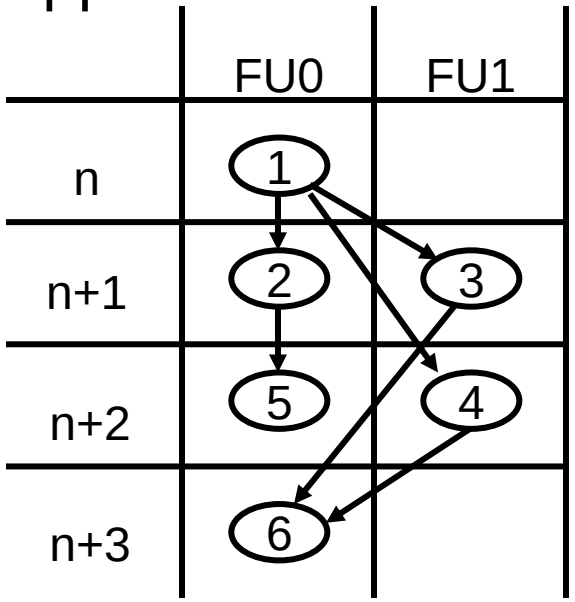
Instruction scheduling

- A process of mapping a series of instructions into execution resources
  - Decides when and where an instruction is executed

- Data dependence graph



- Mapped to two FUs



Instruction scheduling

- A set of wakeup and select operations
  - Wakeup
    - Broadcasts the tags of parent instructions selected
    - Dependent instruction gets matching tags, determines if source operands are ready
    - Resolves true data dependences
  - Select
    - Picks instructions to issue among a pool of ready instructions
    - Resolves resource conflicts
      - Issue bandwidth
      - Limited number of functional units / memory ports

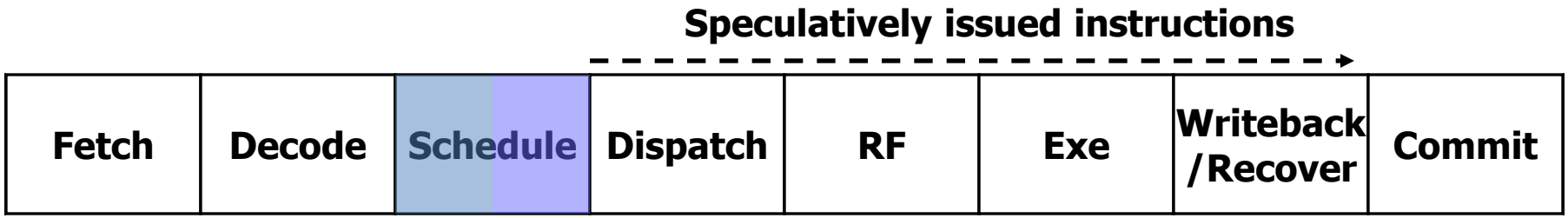
Speculative Scheduling

- Speculatively wakeup dependent instructions even before the parent instruction starts execution
  - Keep the scheduling loop within a single clock cycle
- But, nobody knows what will happen in the future
- Source of uncertainty in instruction scheduling: loads
  - Cache hit / miss, bank conflict
  - Store-to-load aliasing

  ➔ eventually affects timing decisions
- Scheduler assumes that all types of instructions have pre-determined fixed latencies
  - Load instructions are assumed to have a common case (over 90% in general) $DL1 hit latency
  - If incorrect, subsequent (dependent) instructions are replayed

Speculative Scheduling

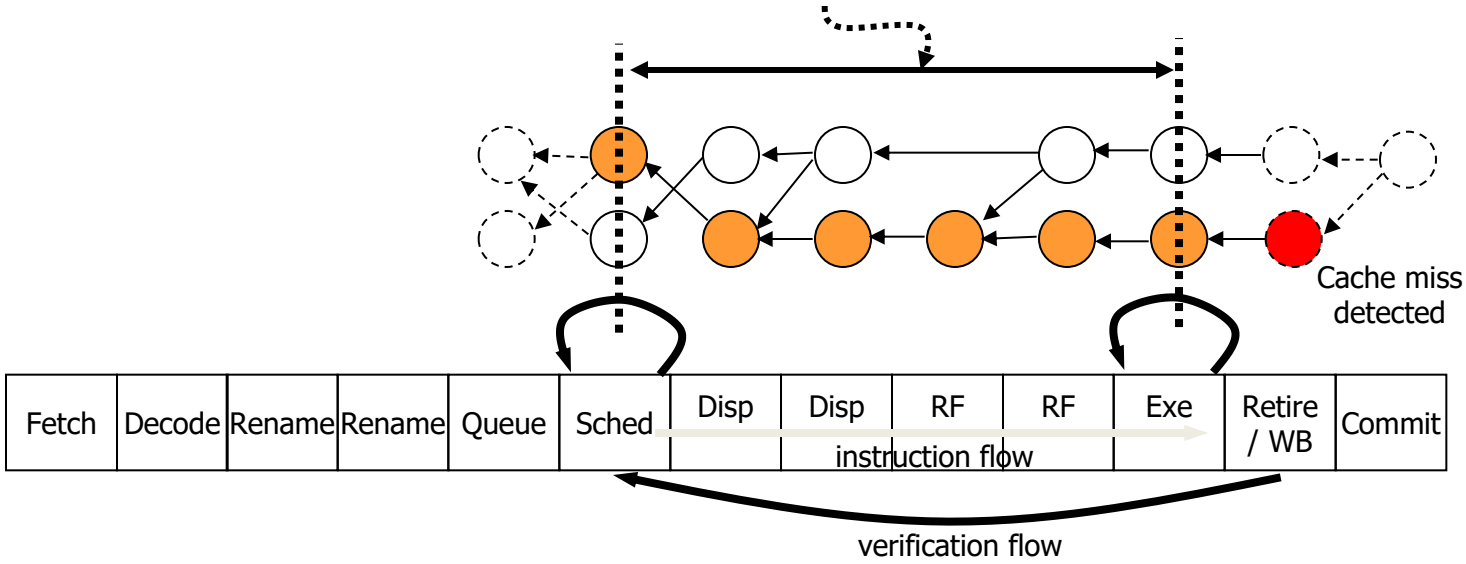
- Overview



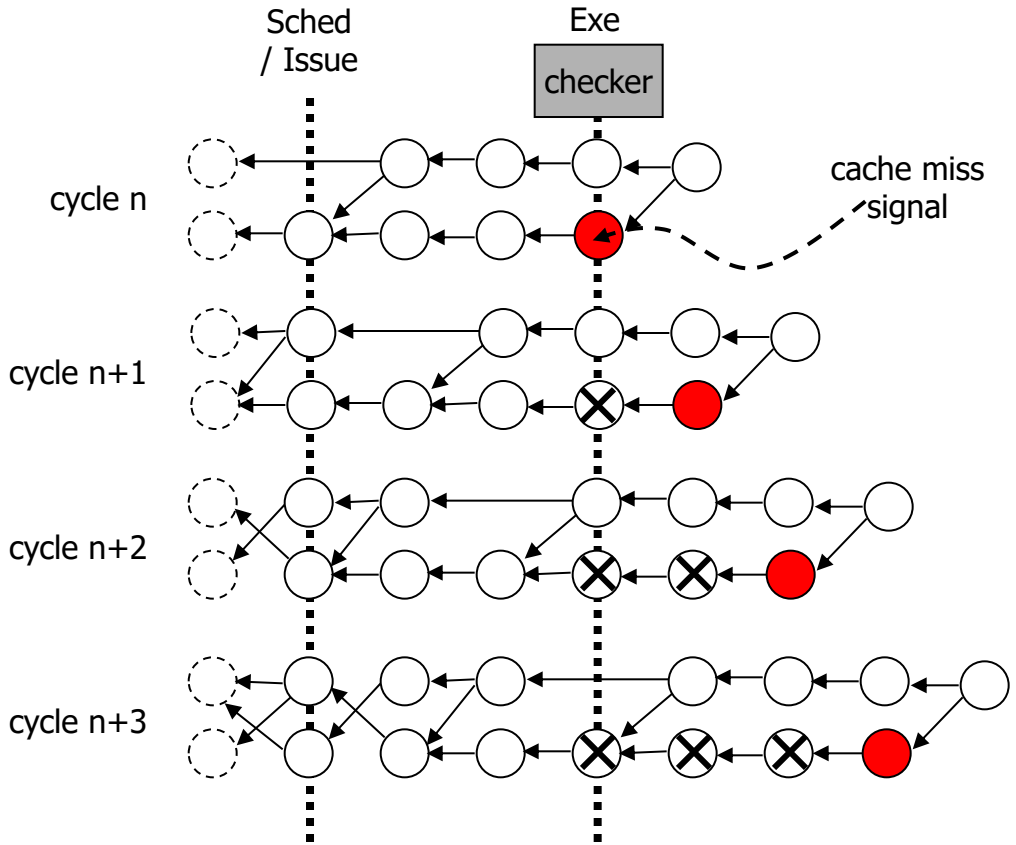
- Unlike the original Tomasulo's algorithm
  - Instructions are scheduled BEFORE actual execution occurs
  - Assumes instructions have pre-determined fixed latencies
    - ALU operations: fixed latency
    - Load operations: assumes $DL1 latency (common case)

Scheduling replay

- Speculation needs verification / recovery
  - There's no free lunch
- If the actual load latency is longer (i.e. cache miss) than what was speculated
  - Best solution (disregarding complexity): replay data-dependent instructions issued under *load shadow*



Issues in scheduling replay



- Cannot stop speculative wavefront propagation
  - Both wavefronts propagate at the same rate
  - Dependent instructions are unnecessarily issued under load misses

Requirements of scheduling replay

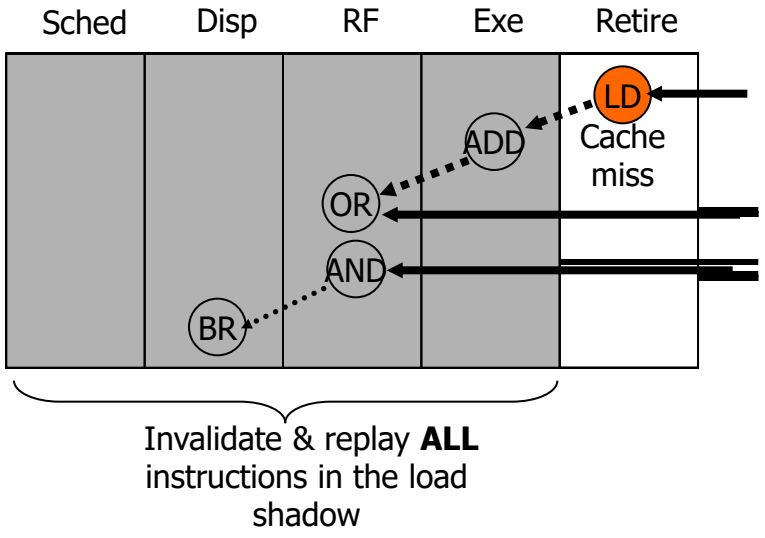
- Propagation of recovery status should be faster than speculative wavefront propagation
- Recovery should be performed on the transitive closure of dependent instructions

- Ideally:
  - All mis-scheduled dependent instructions are invalidated instantly
  - Independent instructions are unaffected

- Simplest Solution: Treat schedule miss as Branch Miss:
  - Flush all subsequent instructions

Scheduling replay schemes

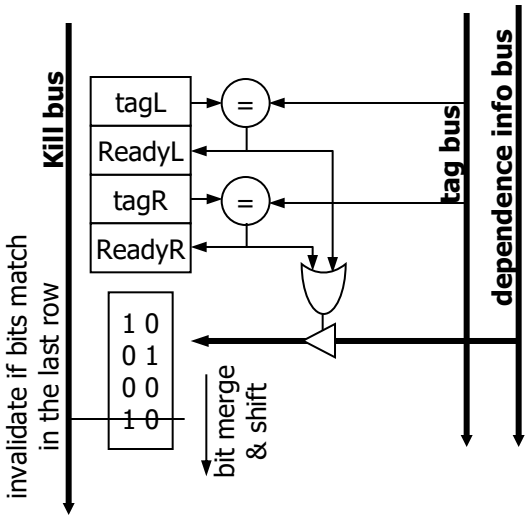
- Alpha 21264: Non-selective replay
  - Replays all dependent and independent instructions issued under load shadow
  - Analogous to squashing recovery in branch misprediction
  - Simple but high performance penalty
    - Independent instructions are unnecessarily replayed

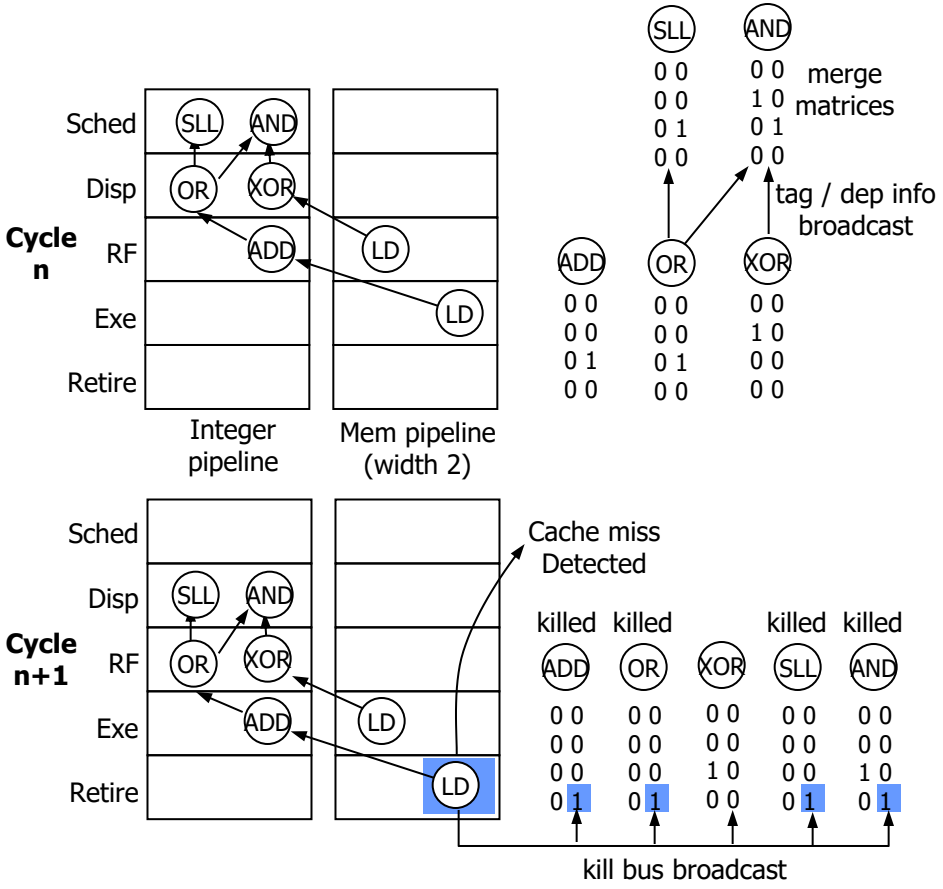


Ideal Scheme

- Each instruction maintains list of dependent Loads
- Mispeculating load broadcasts tag to all in-flight instructions
- (use rename table maintain)

Position-based selective replay





- Ideal selective recovery
  - replay dependent instructions only
- Dependence tracking is managed in a matrix form
  - Column: load issue slot, row: pipeline stages

Limits of Insn-Level Parallelism (ILP)

- Before we build a big superscalar… how much ILP is there?
  - **ILP: instruction-level parallelism** [Fisher`81]
  - Sustainable rate of useful instruction execution
- ILP limit study
  - Assume perfect/infinite hardware, successively add realism
  - Examples: [Wall'88][Wilson+Lam'92]
  - Some surprising results
    - \+ Perfect/infinite "theoretical" ILP: int > 50, FP > 150
      - Sometimes called the **"dataflow limit"**
    - – Real machine "actual" ILP: int ~2, FP ~ 3
    - Fundamental culprits: branch prediction, memory latency
    - Engineering culprits: "window" (RS/SQ/regfile) size, issue width

Limits on Instruction Level Parallelism (ILP)

1970: Flynn

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)

Riseman and Foster's Study

1970: Flynn
1972: Riseman/Foster

- 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						

More that dataflow: Value Prediction

- What is value prediction? Broadly, three salient attributes:
  1. Generate a speculative value (predict)
  2. Consume speculative value (execute)
  3. Verify speculative value (compare/recover)
- This subsumes branch prediction

- Why can value prediction work?
  - Evolution of really sophisticated predictors (e.g. VTAGE)
  - "There's a lot of zeroes out there." (C. Wilkerson)
  - Value locality: "Likelihood of the Likelihood recurrence of a previously-seen value within a storage location"

Some History

- "Classical" value prediction
  - Independently invented by 4 groups in 1995-1996
  1. AMD (Nexgen): L. Widigen and E. Sowadsky, patent filed March 1996, inv. March 1995
  2. Technion: F. Gabbay and A. Mendelson, inv. sometime 1995, TR 11/96, US patent Sep 1997
  3. CMU: M. Lipasti, C. Wilkerson, J. Shen, inv. Oct. 1995, ASPLOS paper submitted March 1996, MICRO June 1996
  4. Wisconsin: Y. Sazeides, J. Smith, Summer 1996

Value Prediction Flop

- Considerable academic interest
  - Dozens of research groups, papers, proposals
- No industry uptake for a long time
  - Intel (x86), IBM (PowerPC), HAL (SPARC) all failed
- Why?
  - Modest performance benefit (< 10%)
  - Power consumption
    - Dynamic power for extra activity
    - Static power (area) for prediction tables
  - Complexity and correctness (risk)
    - Subtle memory ordering issues [MICRO '01]
    - Misprediction recovery [HPCA '04]

- Architects are pursuing ILP once again…

Some Recent Interest

- VTAGE [Perais/Seznec, HPCA 14]
  - Solves many practical problems in the predictor
- EOLE [Perais/Seznec, ISCA 14]
  - Value predicted operands reduce need for OoO
  - Execute some ops early, some late, outside OoO
  - Smaller, faster OoO window
- Load Value Approximation

[San Miguel/Badr/Enright Jerger, MICRO-47][Thwaites et al., PACT 2014]

- DLVP [Sheikh/Cain/Damodaran, MICRO-50]

Unit Summary

- Modern dynamic scheduling must support precise state
  - A software sanity issue, not a performance issue
- Strategy: Writeback → Complete (OoO) + Retire (iO)
- Two basic designs
  - P6: Tomasulo + re-order buffer, copy based register renaming
    ±Precise state is simple, but fast implementations are difficult
  - R10K: implements true register renaming
    ±Easier fast implementations, but precise state is more complex
- Out-of-order memory operations
  - Store queue: conservative load scheduling (iO wrt older stores)
  - Load queue: opportunistic load scheduling (OoO wrt older stores)
  - Intelligent memory scheduling: hybrid

Dynamic Scheduling Summary

- Out-of-order execution: a performance technique
  - Easier/more effective in hardware than software (isn't everything?)
  - Idea: make scheduling transparent to software
- Feature I: Dynamic scheduling (iO $\rightarrow$ OoO)
  - "Performance" piece: re-arrange insns into high-performance order
  - Decode (iO) $\rightarrow$ dispatch (iO) + issue (OoO)
  - Two algorithms: Scoreboard, Tomasulo
- Feature II: Precise state (OoO $\rightarrow$ iO)
  - "Correctness" piece: put insns back into program order
  - Writeback (OoO) $\rightarrow$ complete (OoO) + retire (iO)
  - Two designs: P6, R10K
- Don't forget about memory scheduling

OOO Performance Considerations

- Issue Width (Maximum Performance)
  - Governed by: "Width" of Fetch/Decode/Rename/Dispatch/ Issue/**LSQ**/Reg-File/Wakeup/Commit
- Instruction Window (Scheduling Scope)
  - Governed by: Size of Instruction Queue/Reservation Stations, Reorder Buffer (ROB), Register Files, LSQ
- Dependence Latency (Critical Path)
  - Governed by:
    - Data dependences: Functional Unit Latencies
    - Micro-arch Dependences: Length of critical loops (Fetch Loop, Commit Loop, Scheduler Loop, Branch Resolution Loop)

Where do we put Speculation? (control, memory dependence?)

Where do we put vector-instructions?

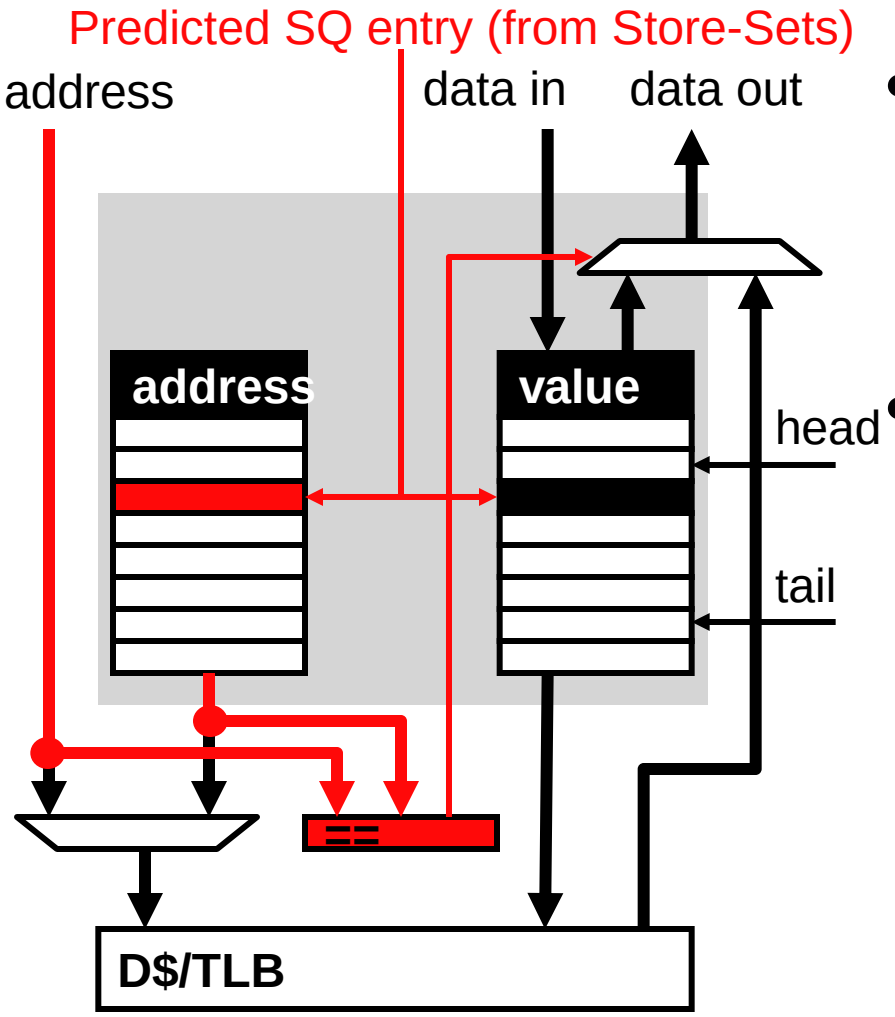
Where do we put Caching?

Class So Far

- Technology:
  - Trends in scaling, implications for architecture constraints
- ISAs:
  - Basic ISA design choices and tradeoffs
- Pipelining and Multi-Issue:
  - IPC vs Frequency Tradeoff, Hazards and Dependences
- Instruction Scheduling:
  - Basic tools of compiler scheduling for in-order machines
  - Mechanics of Dynamic OOO: Tomasulu's, precise state, simple branch prediction, speculation and recovery (P6/R10K)
- Applications:
  - How they affect microarchitecture features

Bonus Slides

Research: Speculative Indexed SQ



- **Observe:** if load forwards, can guess store's SQ position with high accuracy
  - Store-Sets works this way
- **Exploit:** no need to match all stores, use Store-Sets to guess one and match on it
  - CAM+age → RAM+comparator
  - How to verify speculation?
  - Indexed SQ [Sha,Martin,Roth'05]

Clock Rate vs. IPC

- Does frequency vs. width tradeoff actually work?
  - Yes in some places, no in others
  + **Yes**: fetch, decode, rename, retire (all the in-order stages)
  – **No**: issue, execute, complete (all the out-of-order stages)
  - What's the difference?
    - Out-of-order: parallelism doesn't help if insns themselves serial
      - 2 dependent insns execute in 2 cycles, regardless of width
    - In-order: inter-insn parallelism doesn't matter

- Intel Pentium4: **multiple clock domains**
  - In-order stages run at 3.4 GHz, out-of-order stages at 6.8 GHz!
  - Frequency $\propto$ $\text{Power}_{\text{dynamic}}$ $\rightarrow$ high frequency only where necessary

Dynamic Scheduling and Power/Energy

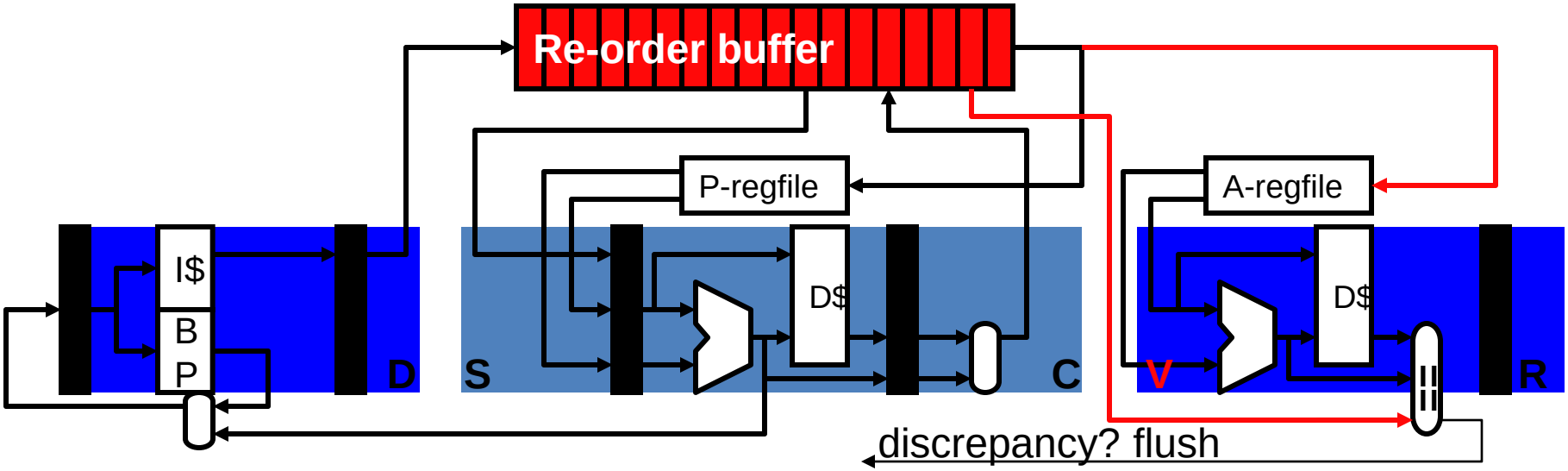
- Is dynamic scheduling low-power?
  - Probably not
  - New SRAMs consume a lot of power
    - Re-order buffer, reservation stations, **physical register file**
  - New CAMs consume even more (relatively)
    - Reservation stations, **load/store queue**
- Is dynamic scheduling low-energy?
  - ± Could be
  - Does performance improvement offset power increase?
  - Are there "deep sleep" modes?

Dynamic Scheduling and Reliability

- How does dynamic scheduling affect reliability?
  - What is the fault model?
  - ± Transient faults (α-particles)? No effect, I guess
  - ± Gradual faults (electro-migration)? Same
  - – Permanent faults (design errors)? Worse, ooo is complicated

- A holistic view of electrical reliability
  - Vulnerability to electrical faults is function of transistor size
  - Mitigate (even eliminate) with larger transistors
  - But larger transistors are slower
  - Overcome clock frequency reductions with increased bandwidth
    - Performance = clock-frequency * IPC
    - Clock-frequency / 2 $\rightarrow$ IPC * 2 $\rightarrow$ superscalar width * 3?

Dynamic Instruction Verification (DIVA)



- Can we tolerate faults in out-of-order (execution) stages?
  - Not directly
  - But can detect them by re-executing insns and comparing results
    - Discrepancy? Flush and restart
  - Insert in-order **verification (V)** stage just before retirement
    - **DIVA** [Austin'99]

DIVA

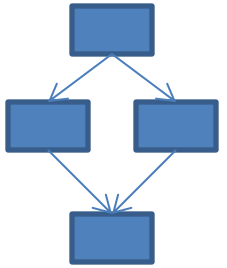
- Why DIVA works
  - Re-execution acts like an in-order stage for parallelization purposes
  - Can re-execute dependent insns in parallel!
  - How come? **"dependence-free checking"**
    - You have original inputs and outputs of all insns
    - Try working this out for yourself
- What DIVA accomplishes
  - \+ Detects transient errors in out-of-order stages
    - Re-execution is parallel → slow clock, big, robust transistors
  - \+ Can also detect design errors
    - Re-execution (in-order) simpler than execution (out-of-order)
    - Less likely to contain rare bugs

Current Dynamic Scheduling Research

- "Critical path modeling"
  - Identify (and optimize) performance critical instructions
- "Scalable schedulers"
  - Support for huge schedulers, several different designs
- "Macro-ops and dataflow mini-graphs"
  - Schedule groups of dependent insns at once (MG: also fetch, retire)
  - Do more with fewer resources
- "Out-of-order fetch and rename"
  - Avoid branch mispredictions by fetching control independent insns
- "WaveScalar"
  - Like an out-of-order Grid processor
  - $$$$
- Much more…

Cool Ideas

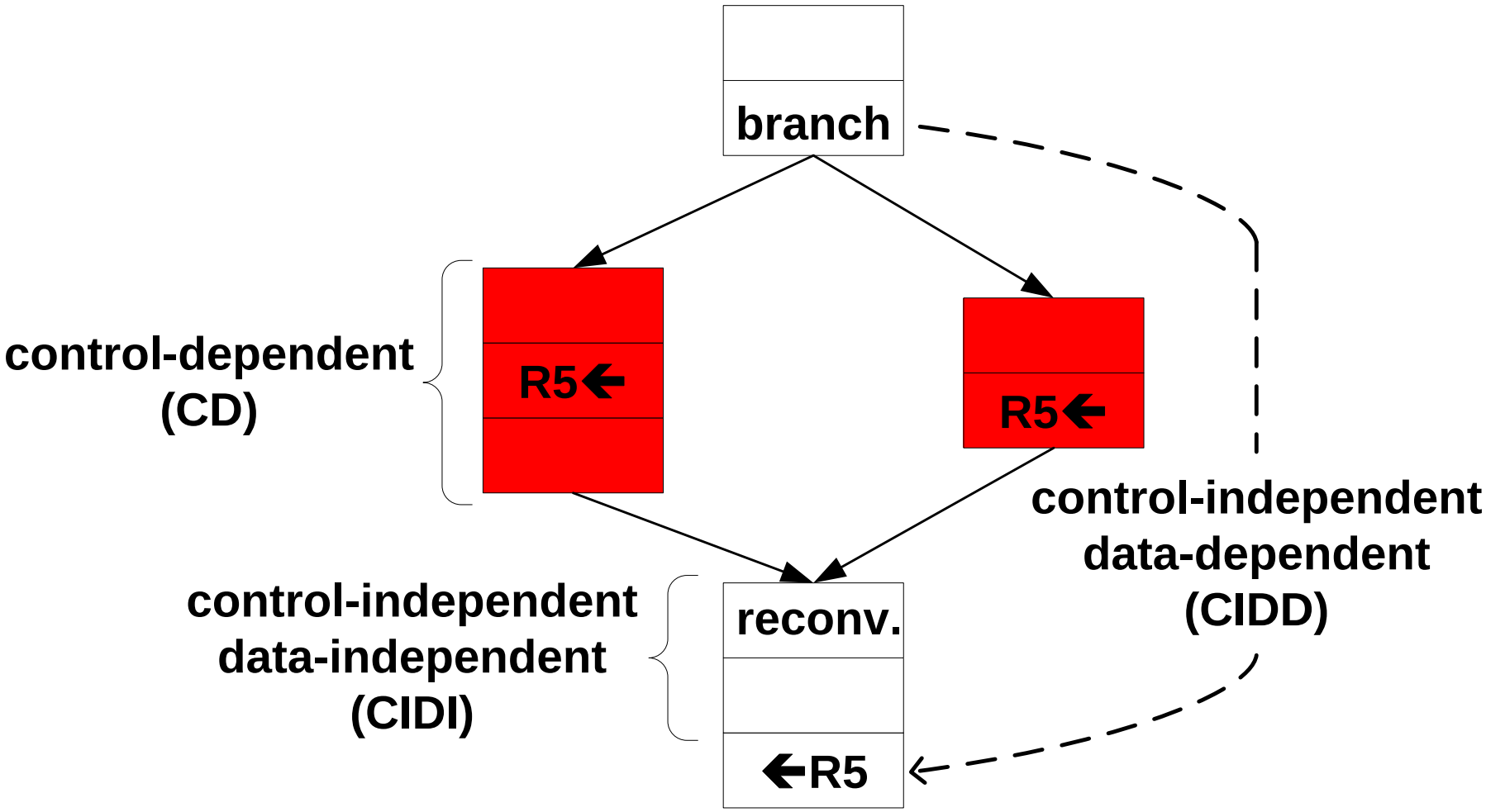
Transparent Control Independence



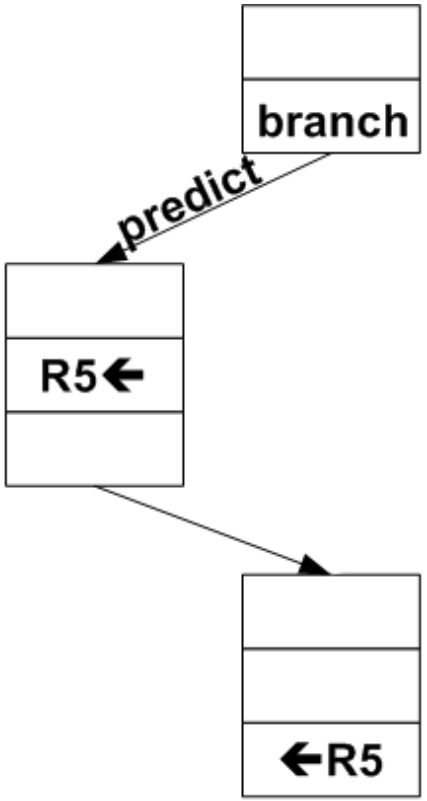
[Al-Zawawi et al., ISCA 07]

- Control flow graph convergence
  - Execution reconverges after branches
  - If-then-else constructs, etc.
- Can we fetch/execute instructions beyond convergence point?
  - Significant potential for ILP shown by limit study [Lam/Wilson, ISCA 92]
- How do we resolve ambiguous register and memory dependences?
- Slides from Al-Zawawi ISCA presentation follow

Control independence basics

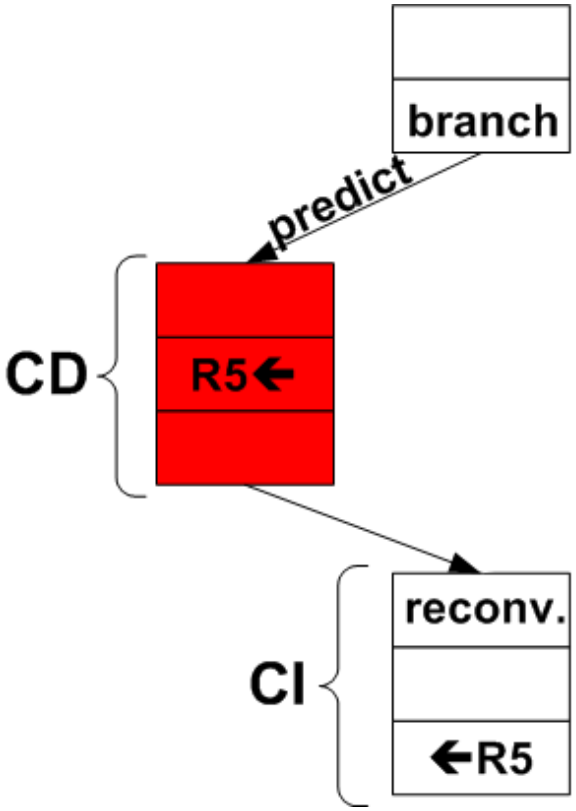


Four steps for exploiting CI



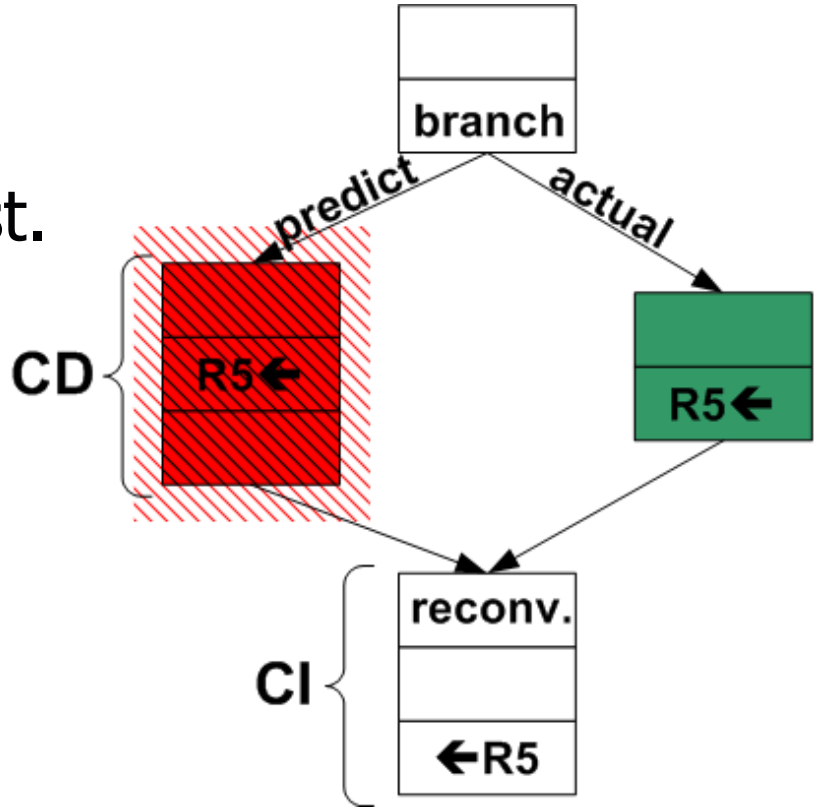
Four steps for exploiting CI

1. Identify reconv. point



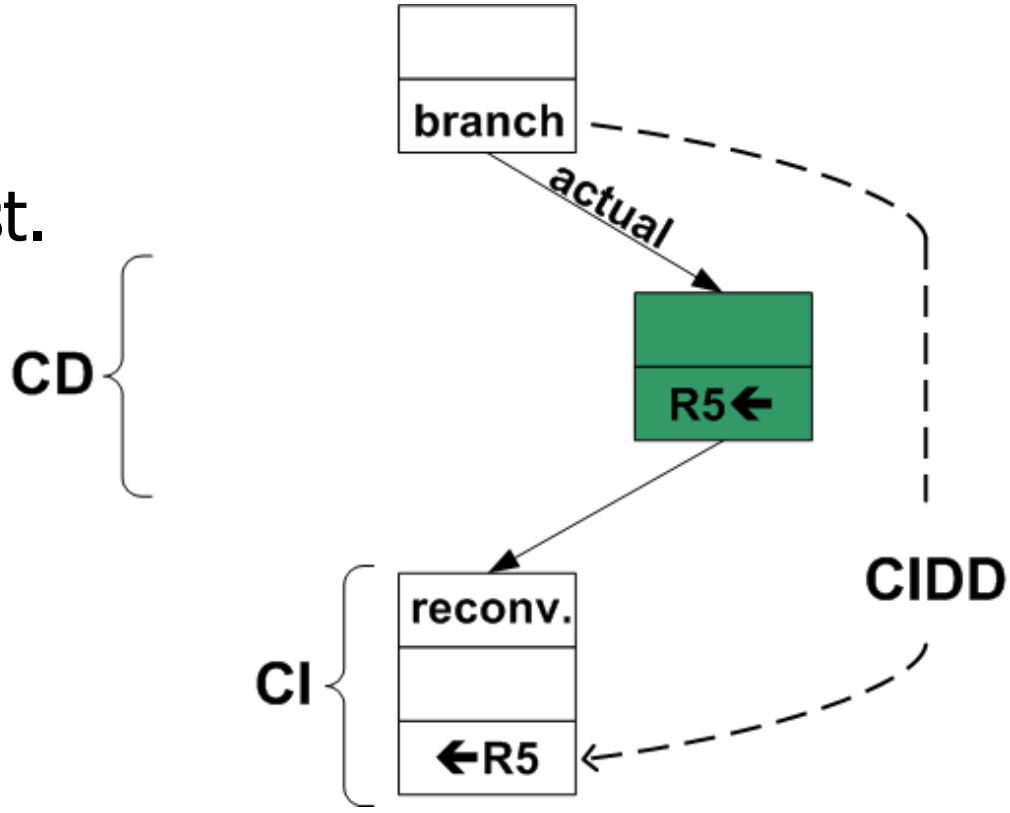
Four steps for exploiting CI

1. Identify reconv. point
2. Remove/Insert CD inst.



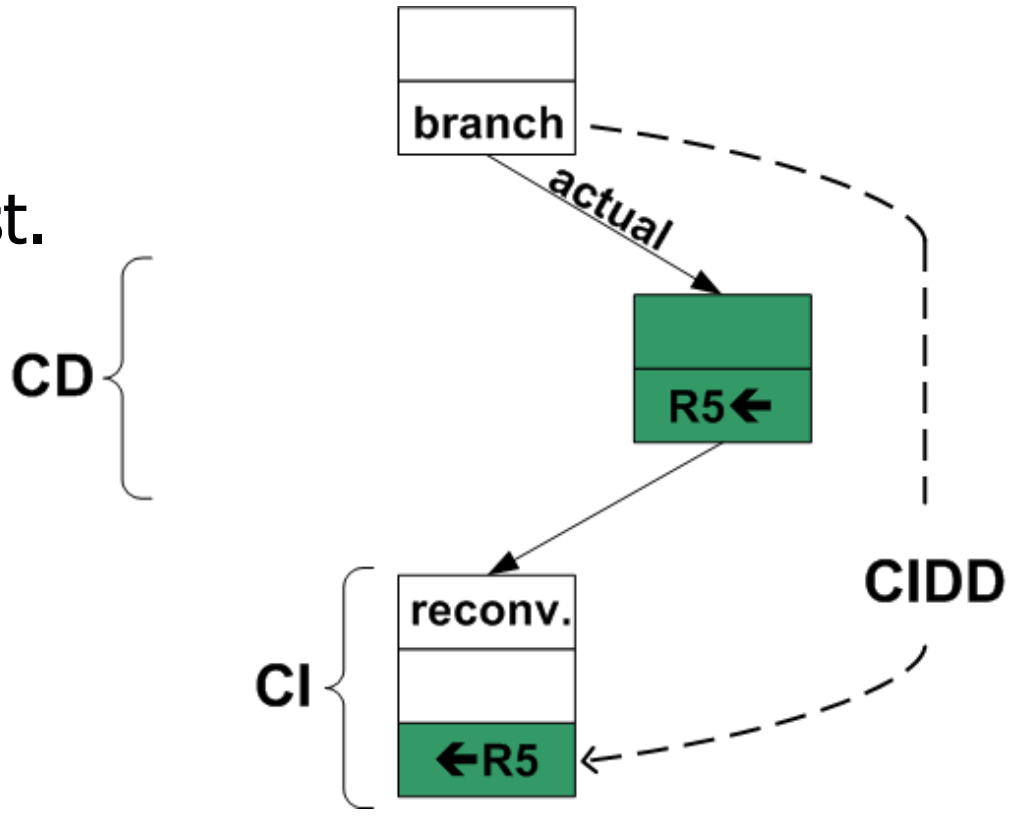
Four steps for exploiting CI

1. Identify reconv. point
2. Remove/Insert CD inst.
3. Identify CIDD inst.

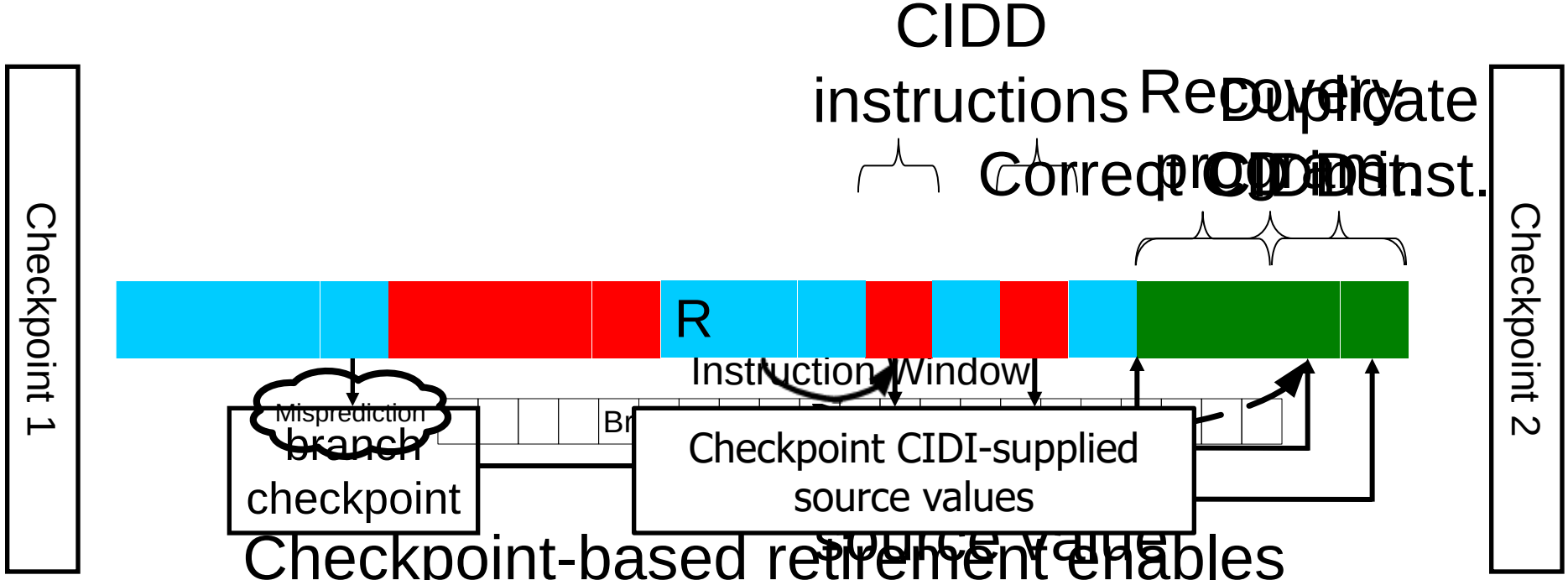


Four steps for exploiting CI

1. Identify reconv. point
2. Remove/Insert CD inst.
3. Identify CIDD inst.
4. Repair CIDD inst.
   a) Fix data dependencies
   b) Re-execute CIDD inst.



TCI misprediction recovery



Transparent Control Independence

TCI repairs program state, not program order

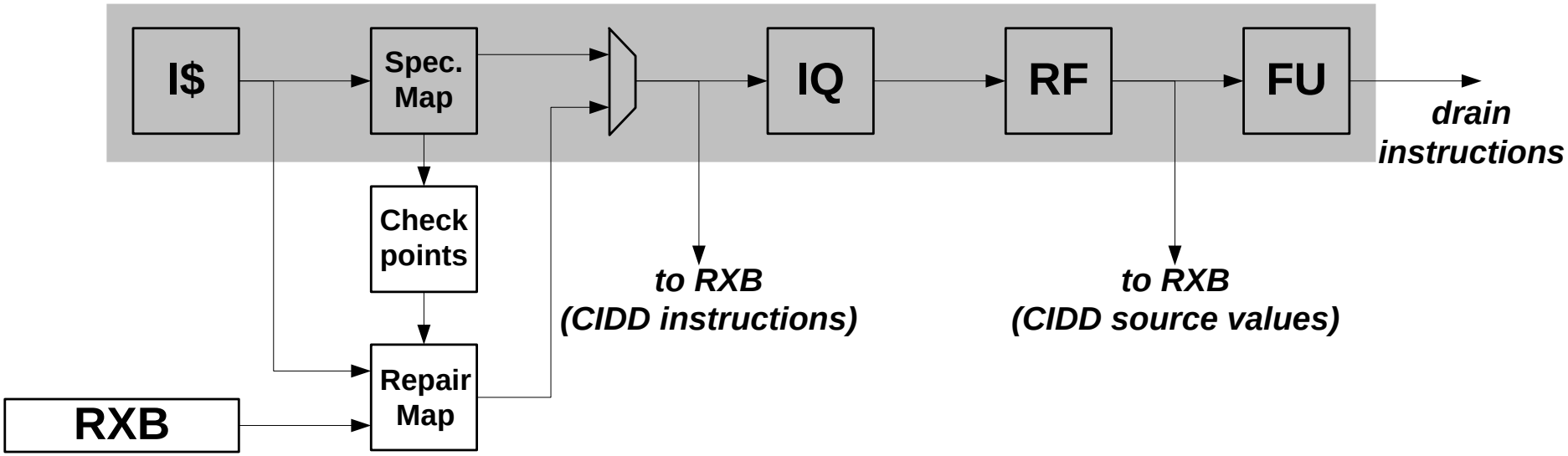
TCI pipeline is recovery-free

- Transparent recovery by fetching additional instructions with checkpointed source values

TCI pipeline is free-flowing

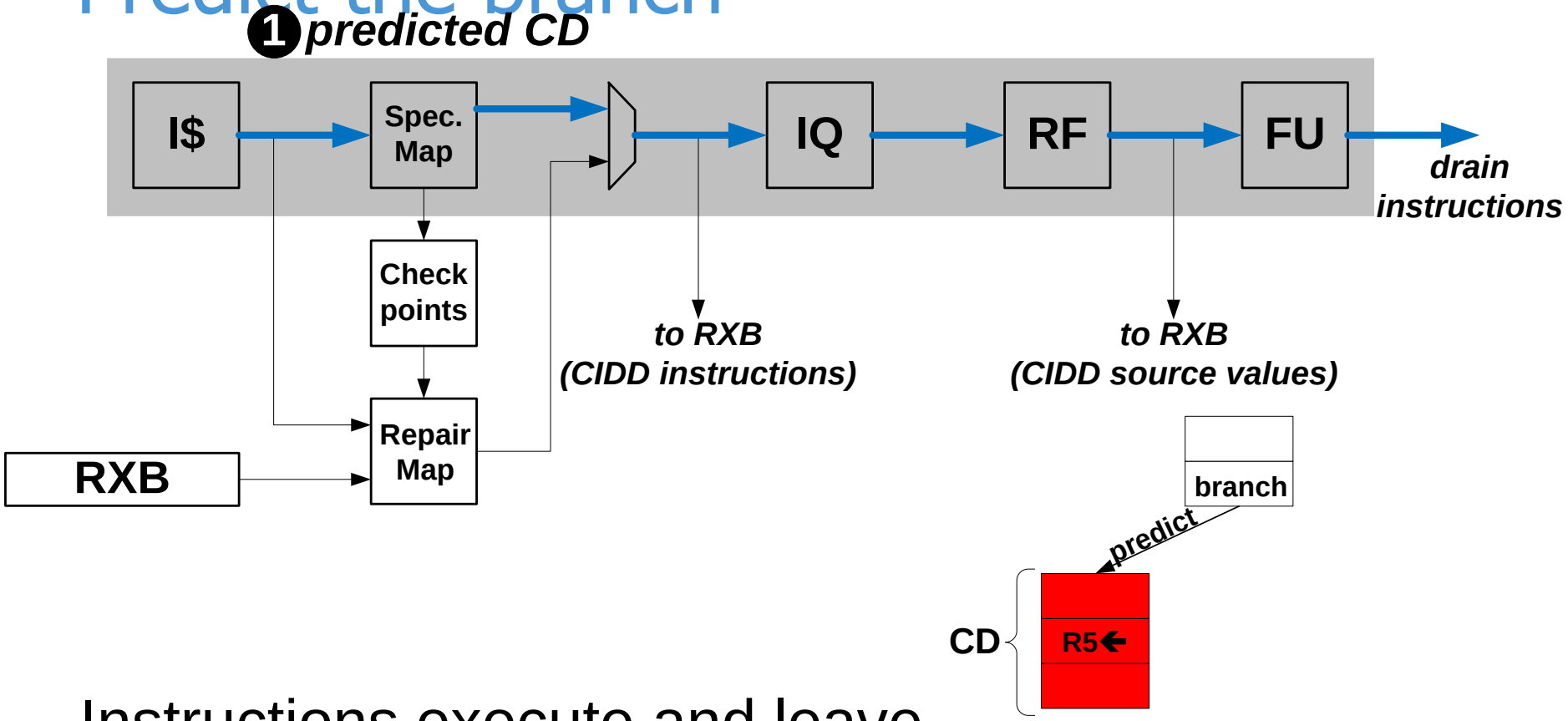
- Leverage conventional speculation to execute correct and incorrect instructions quickly and efficiently
- Completed instructions free their resources

TCI microarchitecture



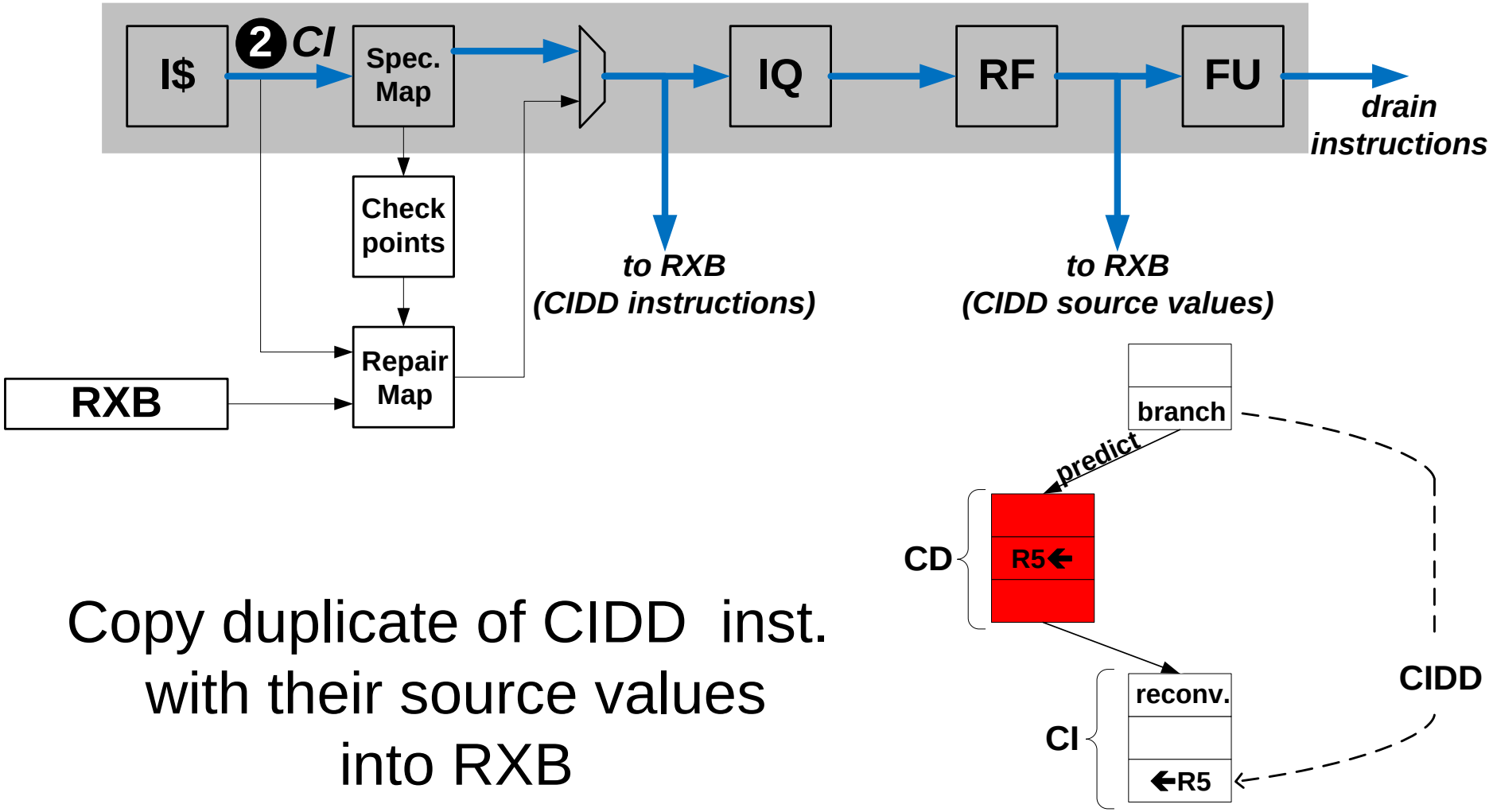
- Add repair rename map
- Add selective re-execution buffer (RXB)

Predict the branch



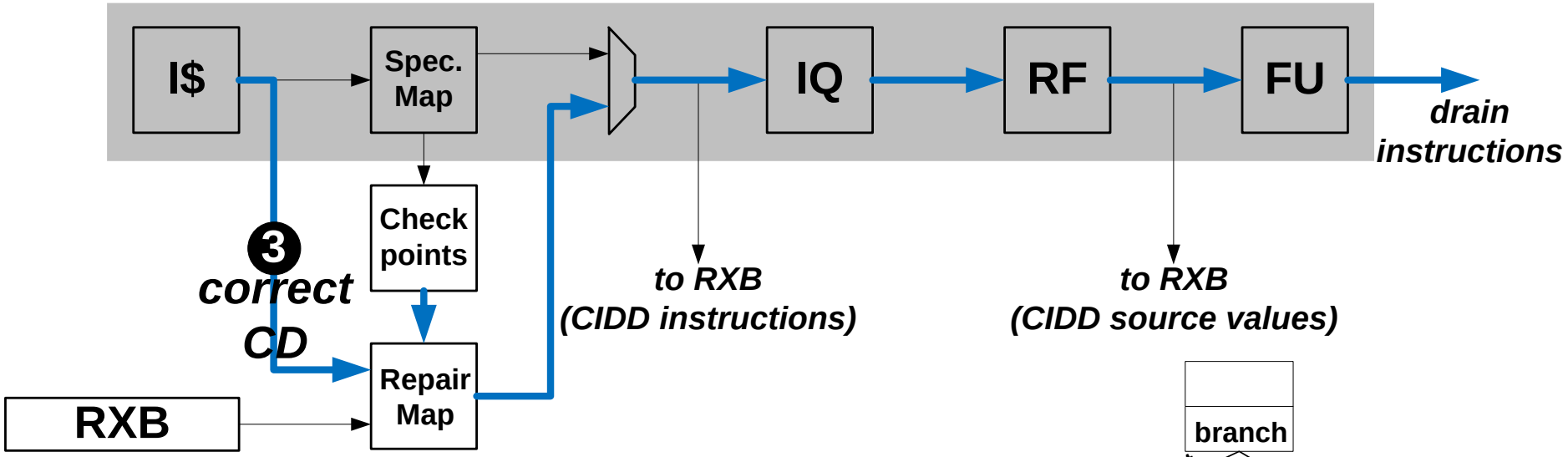
Instructions execute and leave the pipeline when done

Construct recovery program

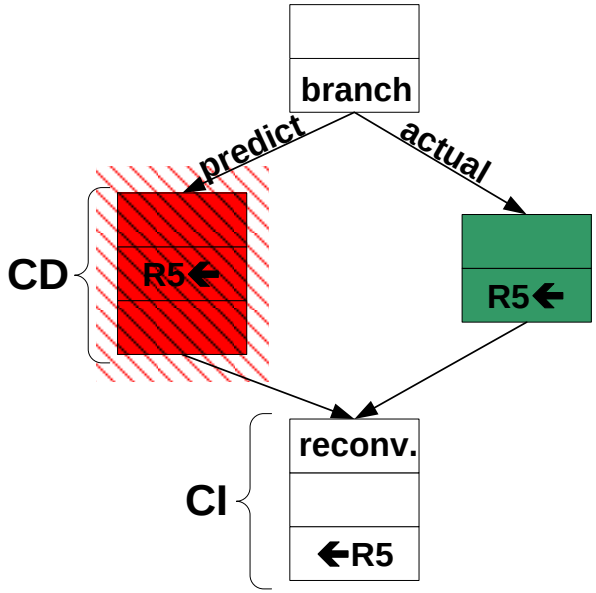


Copy duplicate of CIDD inst. with their source values into RXB

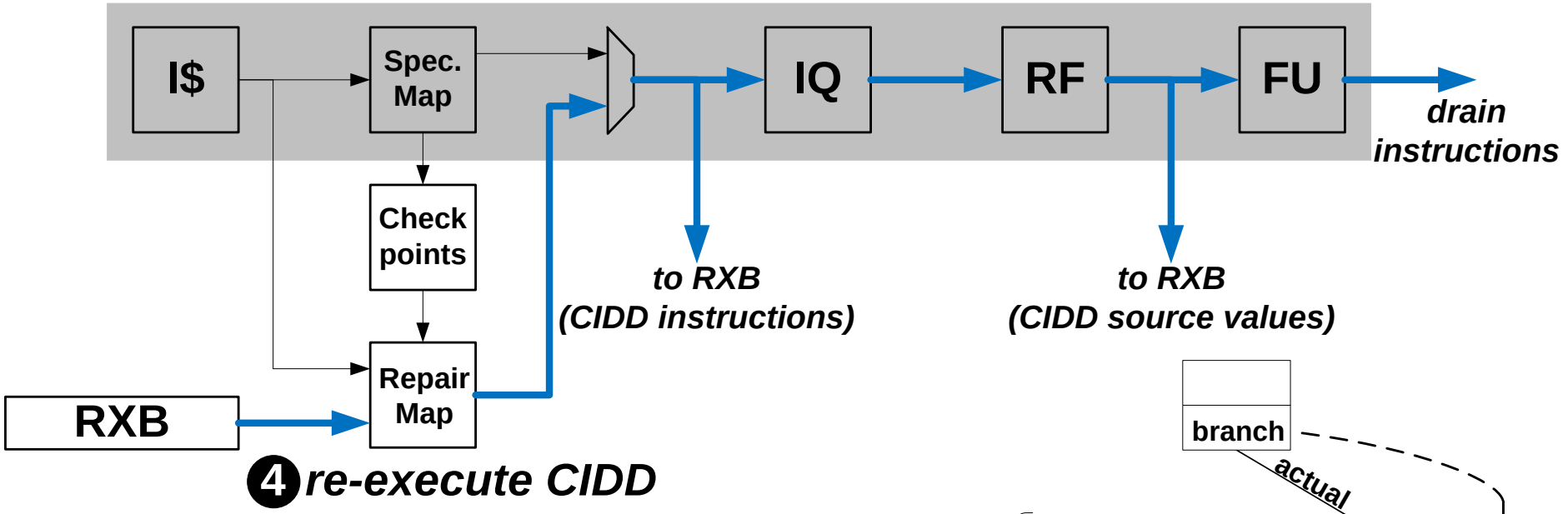
Insert correct CD instructions



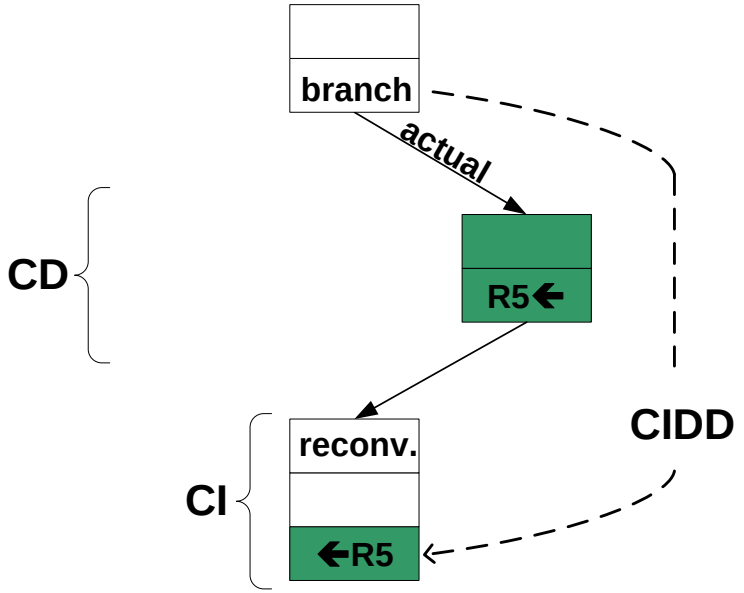
Load branch checkpoint into repair rename map, then fetch correct CD inst.



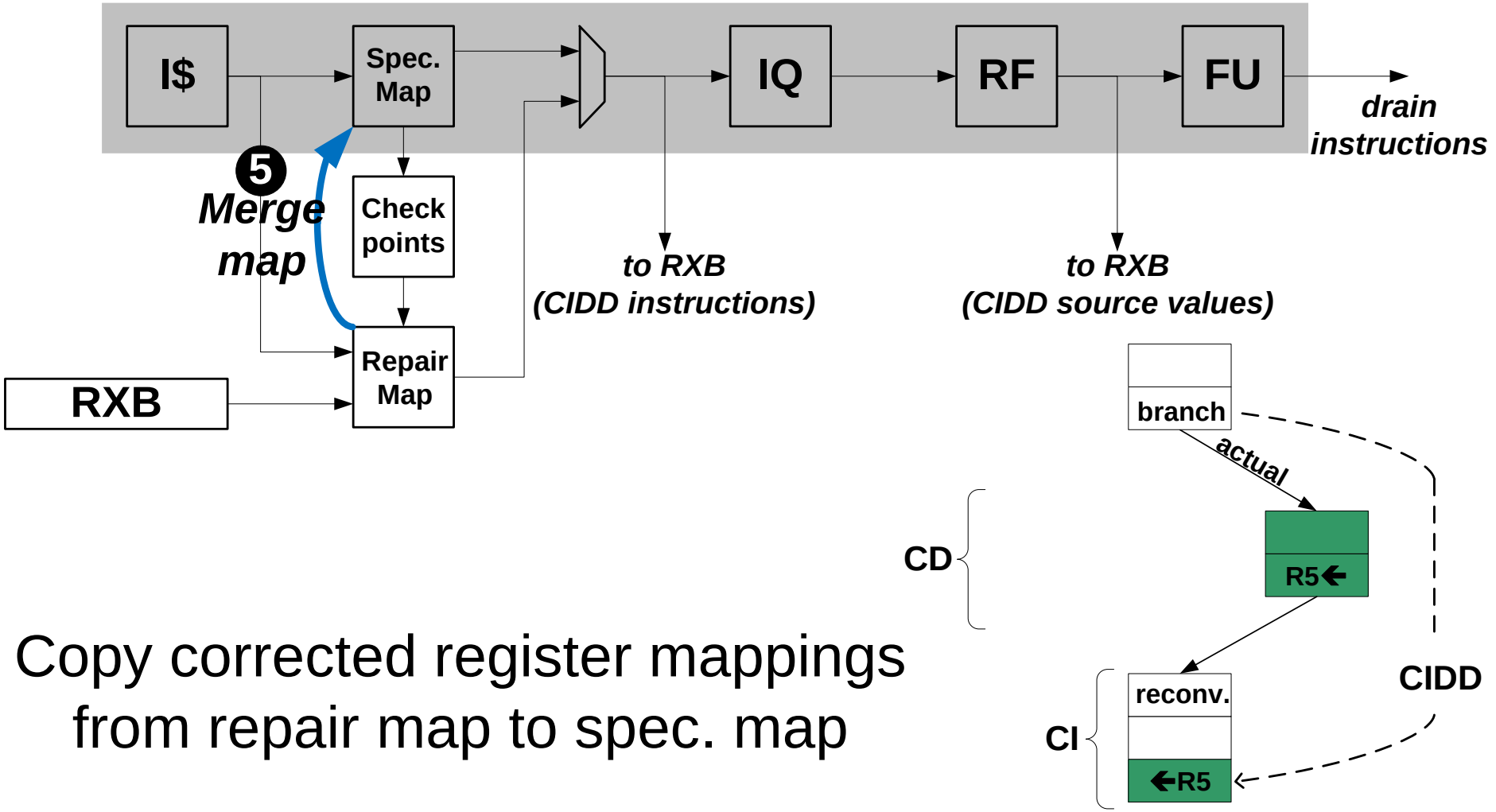
Repair & re-execute CIDD instructions



Inject duplicate CIDD inst. with their checkpointed source values



Merge repair & spec. rename maps



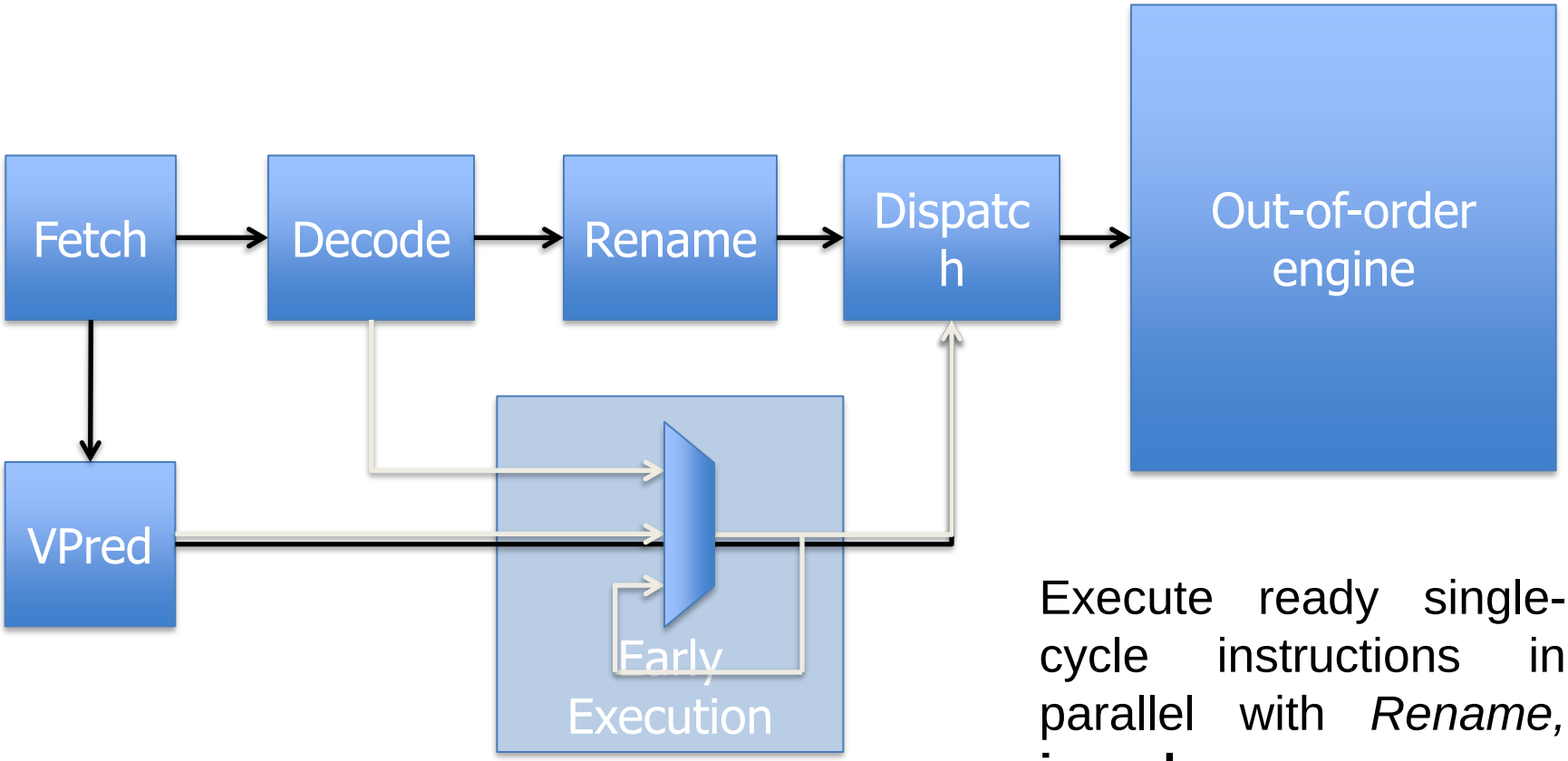
Copy corrected register mappings from repair map to spec. map

Transparent Control Independence

- TCI employs CFP-like slice buffer to reconstruct state
  - Instructions with ambiguous dependences buffered
  - Reinsert them the same way forward load miss slice is reinserted
- "Best" CI proposal to date, but still very complex and expensive, with moderate payback
- Main reason to pursue CI: mispredicted branches
  - This is a moving target
  - Branch misprediction rates have dropped significantly even since 2007

Cool Idea 2: Advanced Value Prediction

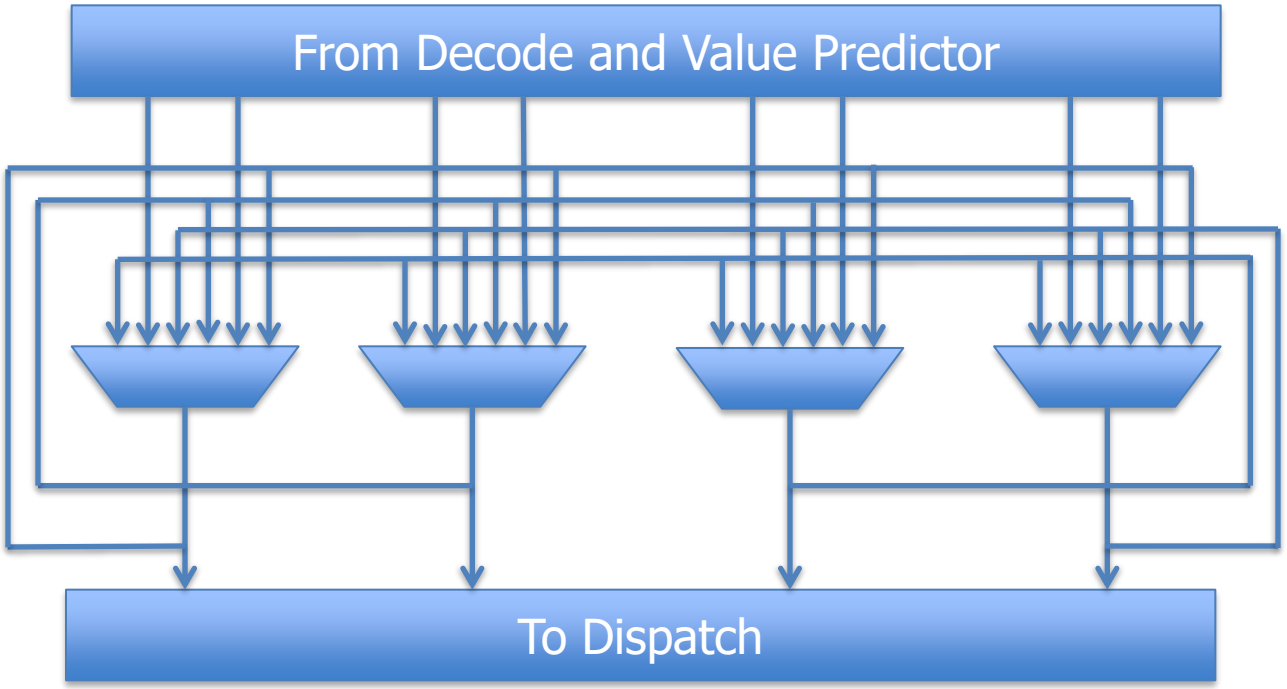
Introducing Early Execution



Execute ready single-cycle instructions in parallel with *Rename*, **in-order.**

Do not dispatch to the IQ.

Early Execution Hardware

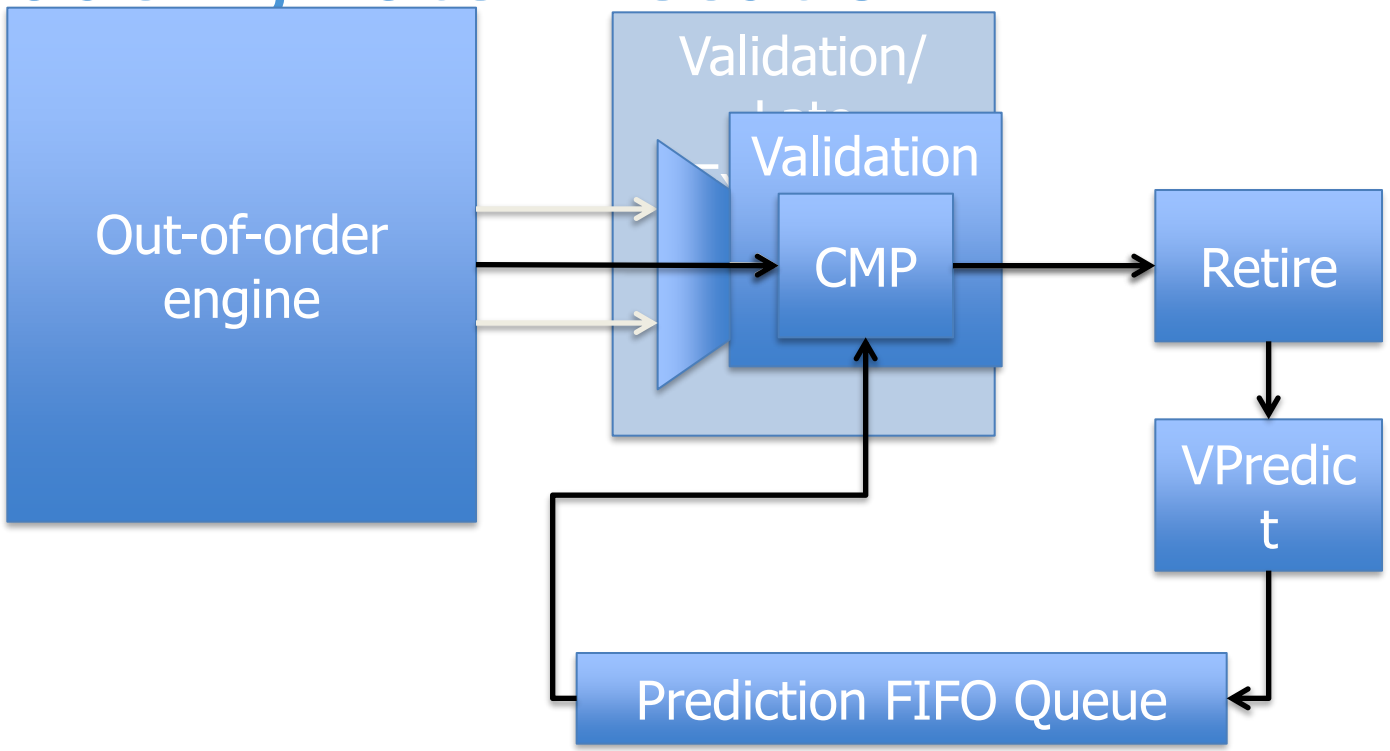


Values come from:

- *Decode* (Immediate)
- Value Predictor
- Bypass Network

- Execute what you can, write in the PRF with the ports provisioned for VP.

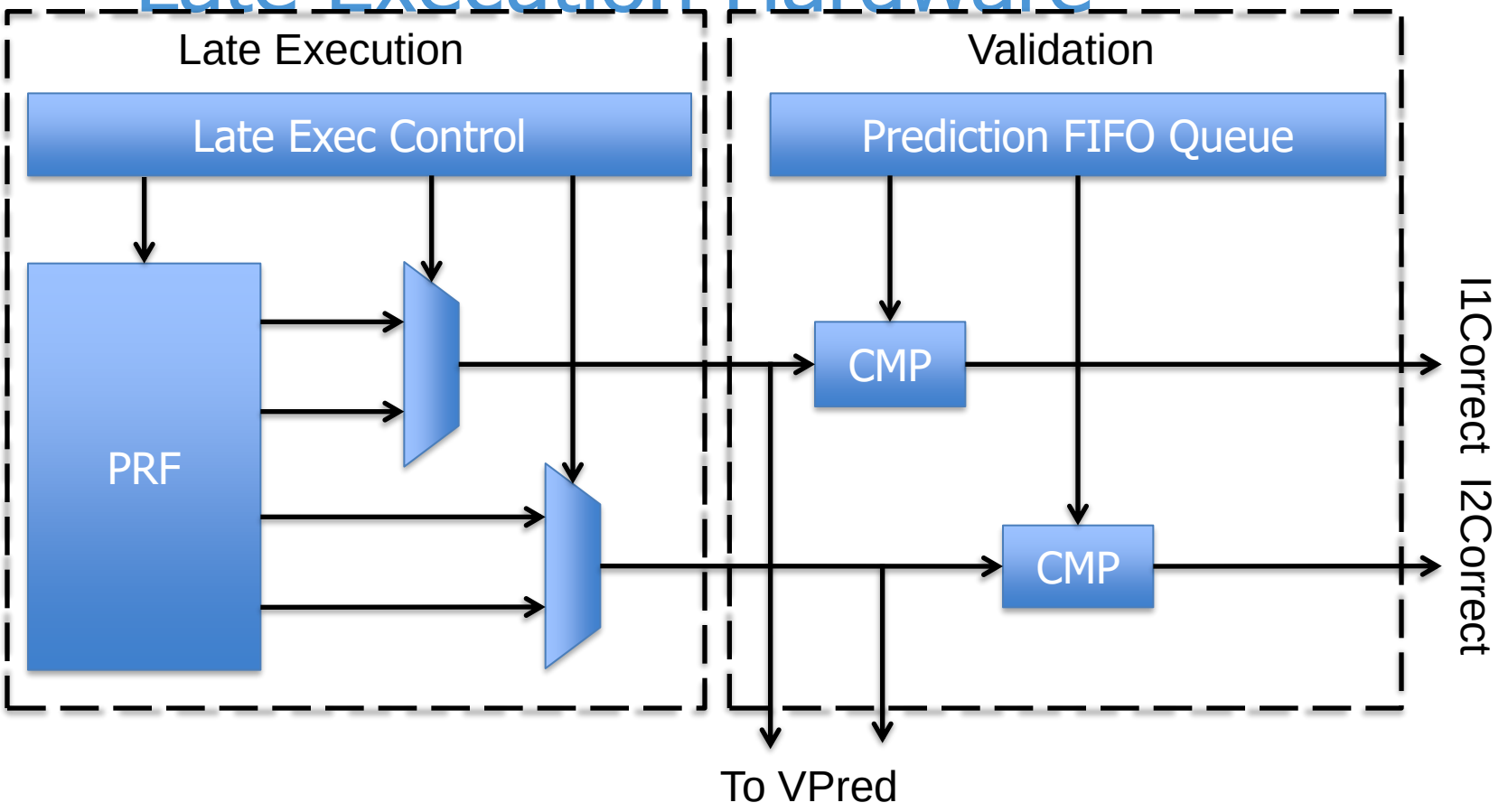
Introducing Late Execution



Execute single-cycle predicted instructions just before retirement, **in-order**.

Do not dispatch to the IQ either.

Late Execution Hardware



- Execute just before validation and retirement by leveraging the ports provisioned for validation.

{Early | OoO | Late} Execution: EOLE

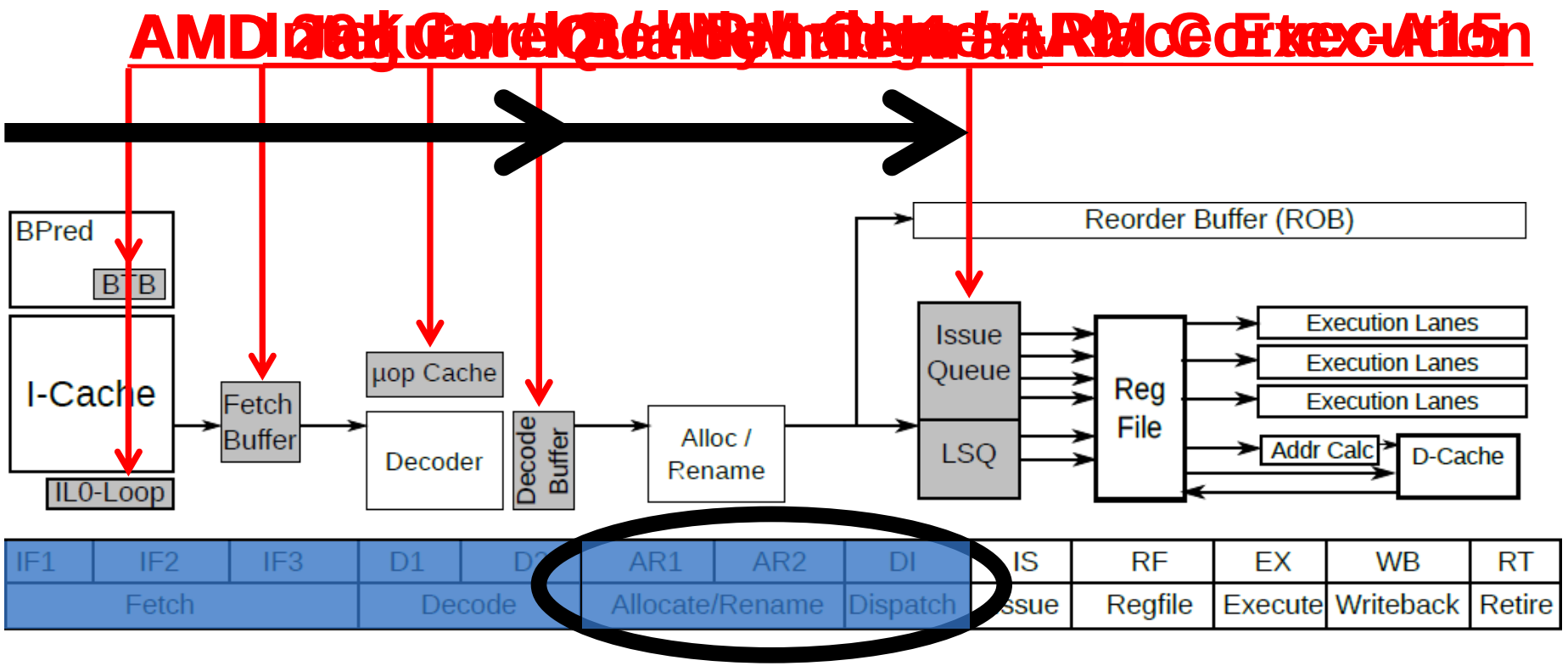
- Much less instructions enter the IQ: We may be able to reduce the issue-width:
  - Simpler IQ.
  - Less ports on the PRF.
  - Less bypass.
  - ➢ Simpler OoO.

- Non critical predictions become useful as the instructions can be late-executed.

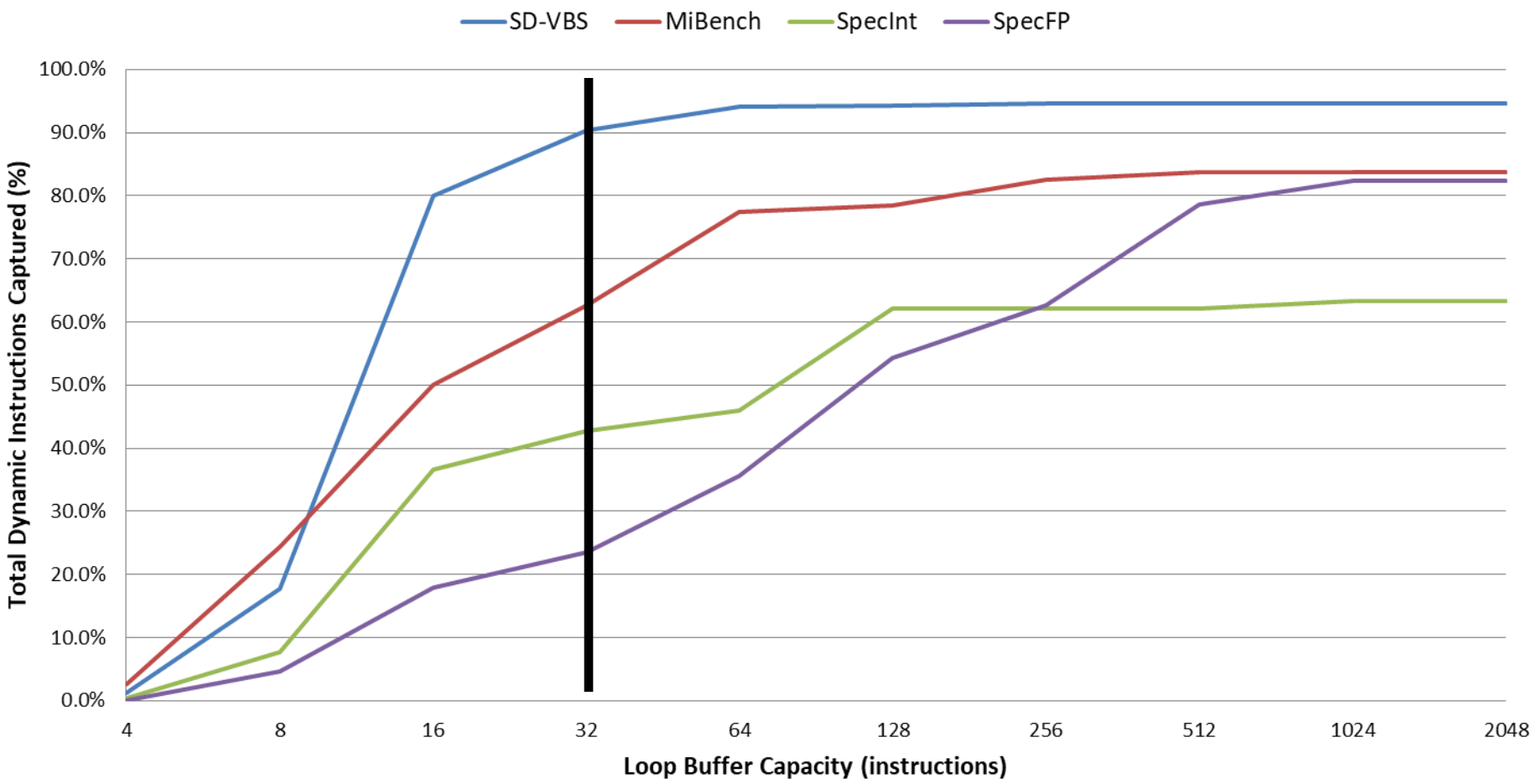
- What about hardware cost?

Cool Idea 3: Loop Buffers

Motivation – Loop Evolution

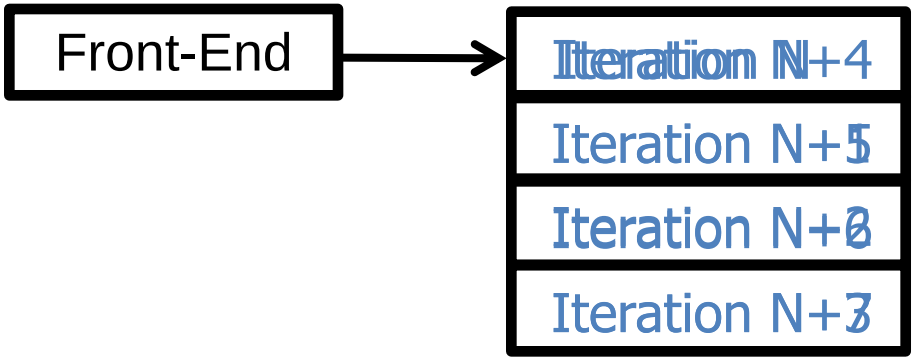


Motivation – Loop Opportunity



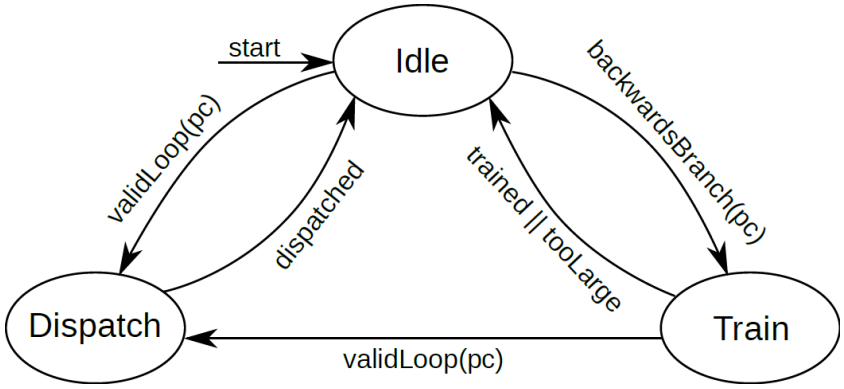
In-Place Loop Execution

- Execute loops in-place
  - Eliminate fetch/branch/dispatch overheads
  - Reuse back-end structures
- Necessary Modifications
  - Loop Detection / Dispatch Logic
  - Dependence Linking
  - Reusable backend structures
    - IQ Entries, LSQ Entries, Physical Registers



Frontend Loop Logic

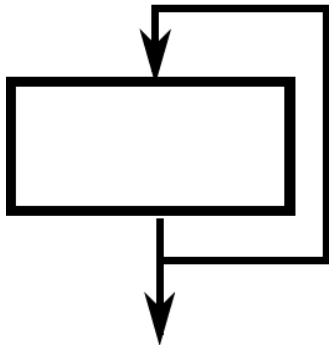
- Primary Responsibilities
  - Identify loops and resource requirements
  - Dispatch loops
  - Incorporate feedback
- Loop Identification
  - Triggered by backwards branch
  - Unlimited control flow
  - Utilizes simple state machine and registers
- Details in [Hayenga, HPCA 2014]



Loop Types

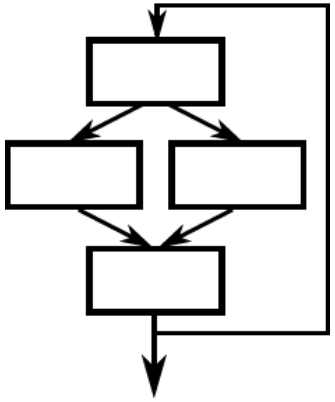
Simple

```
start:  ld r0,[r1, r3]
        str r0, [r2, r3]
        sub r3, r3, #4
        cmp r3,#0
        bne start
```



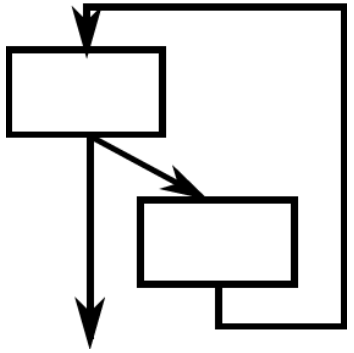
Complex

```
start:  ld r0, [r1, r2]
        cmp r0, #0
        beq skip
        str #0xF, [r1, r2]
skip:   sub r2, r2, #4
        cmp r2, 0
        bne start
```



Early Exit

```
start:  ldr r0, [r1]
        cmp r0, #0
        beq exit
        add r1,r1, #1
        b start
```



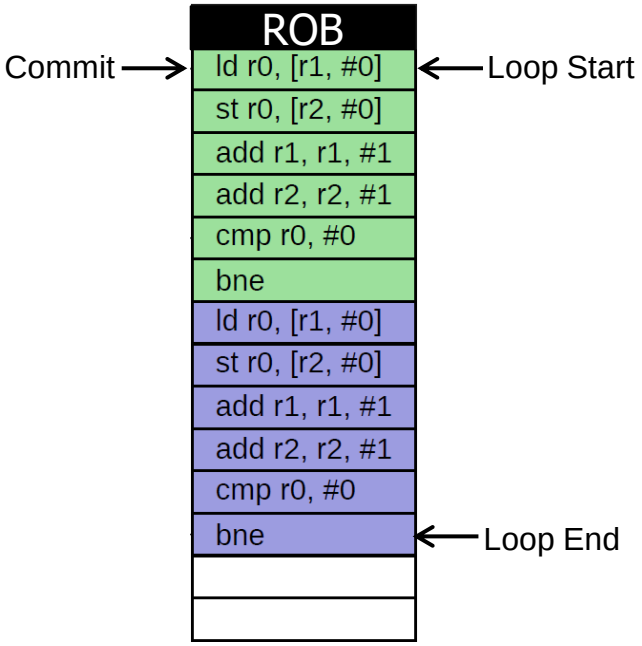
Queue Management

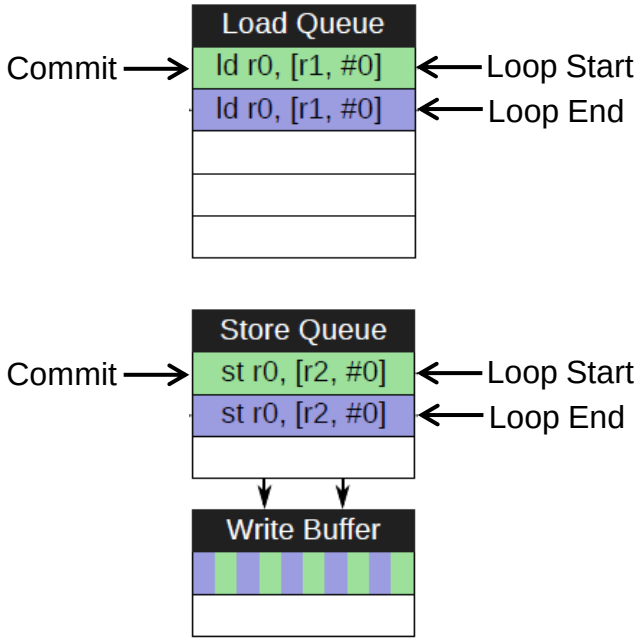
Source Code

```
while(*dst++ = *src++) { }
```

Assembly

```
str_cpy: ld r0, [r1, #0]
         st r0, [r2, #0]
         add r1, r1, #1
         add r2, r2, #1
         cmp r0, #0
         bne str_cpy
```





LSQ – Conventional Ordering

Store Color	I-Stream	
0 1	ld r6, [r7] st r8, [r9]	Before Loop

1
2
	ld r0, [r1, r3]
	st r0, [r2, r3]
	add r3, r3, #1
	cmp r0, #0
	bne
		Iteration #1

2
3
	ld r0, [r1, r3]
	st r0, [r2, r3]
	add r3, r3, #1
	cmp r0, #0
	bne
		Iteration #2

3 4	ld r0, [r7] st r1, [r5]	After Loop

LSQ – Loop Ordering

Store Color	I-Stream	
0 1	ld r6, [r7] st r8, [r9]	Before Loop

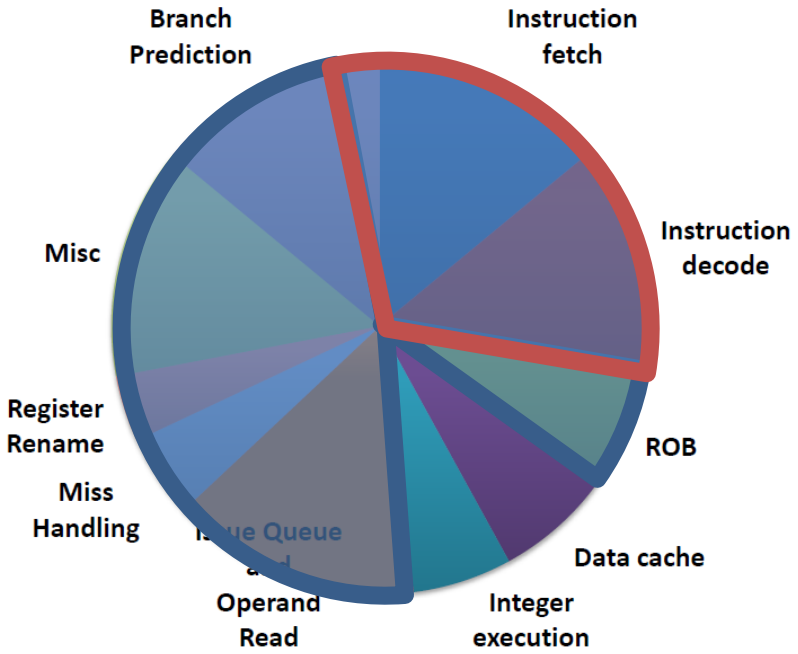
1
2
	ld r0, [r1, r3]
	st r0, [r2, r3]
	add r3, r3, #1
	cmp r0, #0
	bne
		Iteration #1

1
2
	ld r0, [r1, r3]
	st r0, [r2, r3]
	add r3, r3, #1
	cmp r0, #0
	bne
		Iteration #2

3 4	ld r0, [r7] st r1, [r5]	After Loop

In-place Loop Cache Benefit

ARM Cortex A15 [Source: NVIDIA]



- On average 20% fewer instructions fetched
- Still significant opportunity remaining