CSC 631: High-Performance Computer Architecture

Spring 2017
Lecture 6: Out-of-Order Processors

Supercomputers

Definitions of a supercomputer:

- Fastest machine in world at given task
- A device to turn a compute-bound problem into an I/O bound problem
- Any machine costing $30M+
- Any machine designed by Seymour Cray

- CDC6600 (Cray, 1964) regarded as first supercomputer

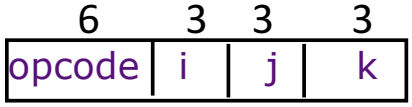
CDC 6600 *Seymour Cray, 1963*

- A fast pipelined machine with 60-bit words
  - 128 Kword main memory capacity, 32 banks
- Ten functional units (parallel, unpipelined)
  - Floating Point: adder, 2 multipliers, divider
  - Integer: adder, 2 incrementers, ...
- Hardwired control (no microcoding)
- *Scoreboard* for dynamic scheduling of instructions
- Ten Peripheral Processors for Input/Output
  - a fast multi-threaded 12-bit integer ALU
- Very fast clock, 10 MHz (FP add in 4 clocks)
- >400,000 transistors, 750 sq. ft., 5 tons, 150 kW, novel freon-based technology for cooling
- Fastest machine in world for 5 years (until 7600)
  - over 100 sold ($7-10M each)

CDC 6600: A Load/Store Architecture

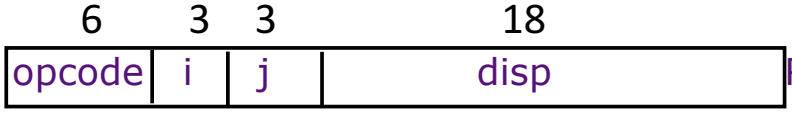
- Separate instructions to manipulate three types of reg.
  - 8x60-bit data registers (X)
  - 8x18-bit address registers (A)
  - 8x18-bit index registers (B)

- All arithmetic and logic instructions are register-to-register



Ri ← Rj op Rk

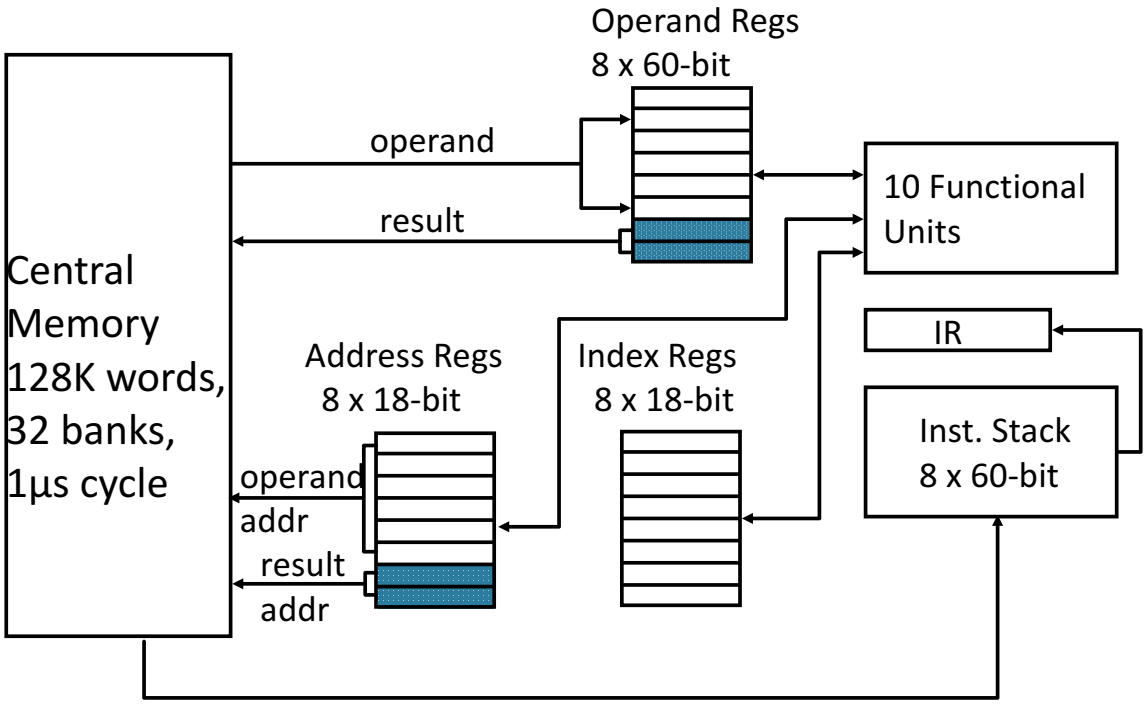
- Only Load and Store instructions refer to memory!



Ri ← M[Rj + disp]

Touching address registers 1 to 5 initiates a load
6 to 7 initiates a store
- very useful for vector operations

CDC 6600: Datapath



CDC6600 ISA designed to simplify high-performance implementation

- Use of three-address, register-register ALU instructions simplifies pipelined implementation
  - Only 3-bit register specifier fields checked for dependencies
  - No implicit dependencies between inputs and outputs
- Decoupling setting of address register (Ar) from retrieving value from data register (Xr) simplifies providing multiple outstanding memory accesses
  - Software can schedule load of address register before use of value
  - Can interleave independent instructions inbetween
- CDC6600 has multiple parallel but unpipelined functional units
  - E.g., 2 separate multipliers
- Follow-on machine CDC7600 used pipelined functional units
  - Foreshadows later RISC designs

CDC6600: Vector Addition

```
        B0 ← - n
loop:   JZE  B0, exit
        A0 ← B0 + a0        load X0
        A1 ← B0 + b0        load X1
        X6 ← X0 + X1
        A6 ← B0 + c0        store X6
        B0 ← B0 + 1
        jump loop
```

Ai = address register
Bi = index register
Xi = data register

CDC6600 Scoreboard

- Instructions dispatched in-order to functional units provided no structural hazard or WAW
  - Stall on structural hazard, no functional units available
  - Only one pending write to any register
- Instructions wait for input operands (RAW hazards) before execution
  - Can execute out-of-order
- Instructions wait for output register to be read by preceding instructions (WAR)
  - Result held in functional unit until register free

MEMORANDUM

August 28, 1963

Memorandum To: Messrs. A. L. Williams
T. V. Learson
H. W. Miller, Jr.
E. R. Piore
O. M. Scott
M. B. Smith
A. K. Watson

Last week CDC had a press conference during which they officially announced their 6600 system. I understand that in the laboratory developing this system there are only 34 people, "including the janitor." Of these, 14 are engineers and 4 are programmers, and only one person has a Ph. D., a relatively junior programmer. To the outsider, the laboratory appeared to be cost conscious, hard working and highly motivated.

Contrasting this modest effort with our own vast development activities, I fail to understand why we have lost our industry leadership position by letting someone else offer the world's most powerful computer. At Jenny Lake, I think top priority should be given to a discussion as to what we are doing wrong and how we should go about changing it immediately.

TJW, Jr:jmc

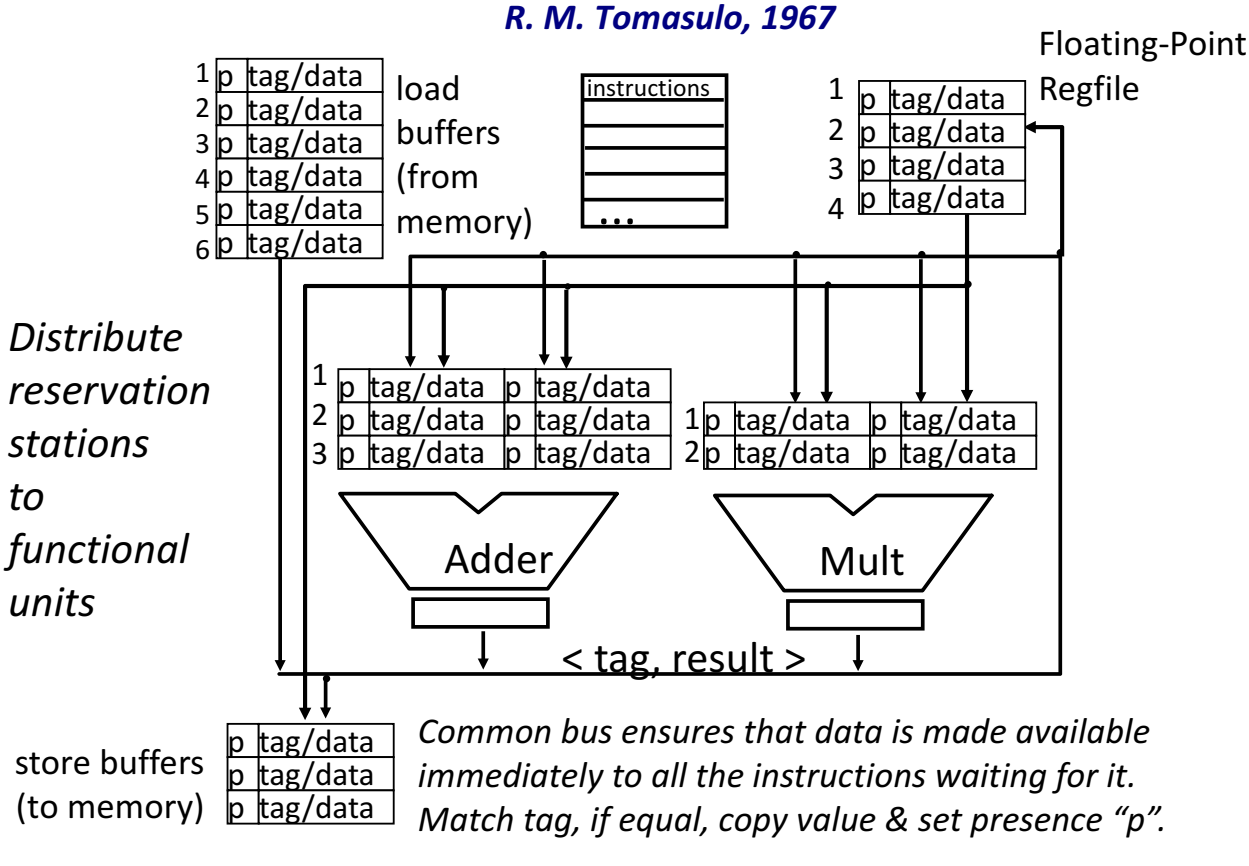
T. J. Watson, Jr.

cc: Mr. W. B. McWhirter

[© IBM]

IBM 360/91 Floating-Point Unit

R. M. Tomasulo, 1967



Common bus ensures that data is made available immediately to all the instructions waiting for it. Match tag, if equal, copy value & set presence "p".

Out-of-Order Fades into Background

Out-of-order processing implemented commercially in 1960s, but disappeared again until 1990s as two major problems had to be solved:

- Precise traps
  - Imprecise traps complicate debugging and OS code
  - Note, precise interrupts are relatively easy to provide
- Branch prediction
  - Amount of exploitable instruction-level parallelism (ILP) limited by control hazards

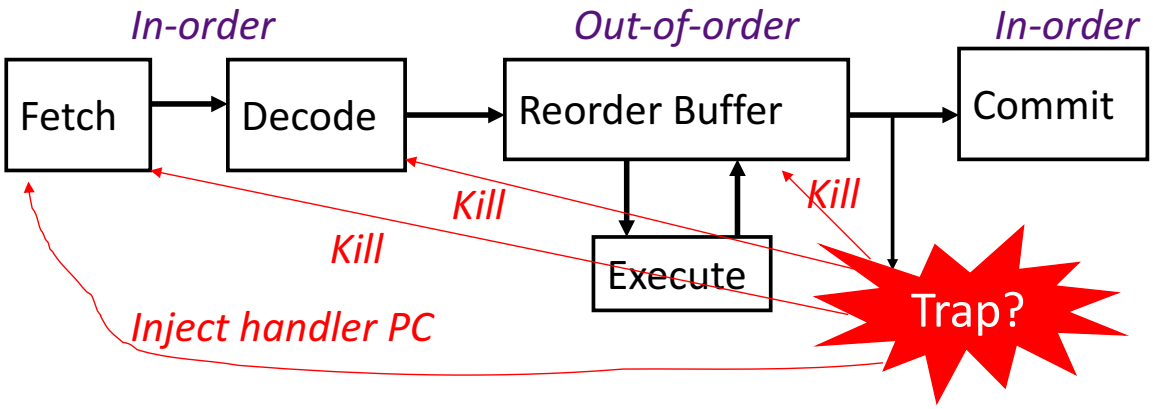
Also, simpler machine designs in new technology beat complicated machines in old technology

- Big advantage to fit processor & caches on one chip
- Microprocessors had era of 1%/week performance scaling

Separating Completion from Commit

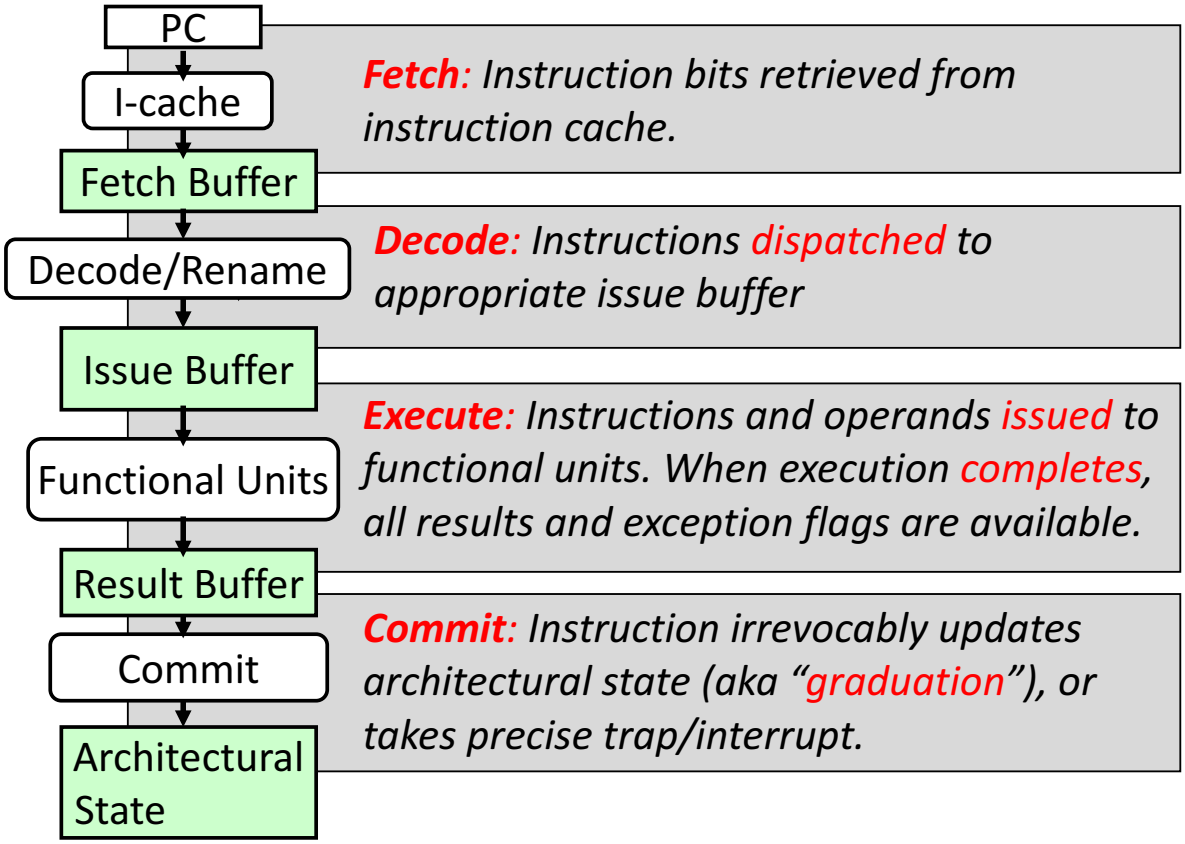
- Re-order buffer holds register results from completion until commit
  - Entries allocated in program order during decode
  - Buffers completed values and exception state until in-order commit point
  - Completed values can be used by dependents before committed (bypassing)
  - Each entry holds program counter, instruction type, destination register specifier and value if any, and exception status (info often compressed to save hardware)
- Memory reordering needs special data structures
  - Speculative store address and data buffers
  - Speculative load address and data buffers

In-Order Commit for Precise Traps



- In-order instruction fetch and decode, and dispatch to reservation stations inside reorder buffer
- Instructions issue from reservation stations out-of-order
- Out-of-order completion, values stored in temporary buffers
- Commit is in-order, checks for traps, and if none updates architectural state

Phases of Instruction Execution



In-Order versus Out-of-Order Phases

- Instruction fetch/decode/rename always in-order
  - Need to parse ISA sequentially to get correct semantics
  - Proposals for speculative OoO instruction fetch, e.g., Multiscalar. Predict control flow and data dependencies across *sequential* program segments fetched/decoded/executed in *parallel*, fixup if prediction wrong
- Dispatch (place instruction into machine buffers to wait for issue) also always in-order
  - Some use "Dispatch" to mean issue, but not in these lectures

In-Order Versus Out-of-Order Issue

- In-order issue:
  - Issue stalls on RAW dependencies or structural hazards, or possibly WAR/WAW hazards
  - Instruction cannot issue to execution units unless all preceding instructions have issued to execution units
- Out-of-order issue:
  - Instructions dispatched in program order to *reservation stations (or other forms of instruction buffer)* to wait for operands to arrive, or other hazards to clear
  - While earlier instructions wait in issue buffers, following instructions can be dispatched and issued out-of-order

In-Order versus Out-of-Order Completion

- All but the simplest machines have out-of-order completion, due to different latencies of functional units and desire to bypass values as soon as available
- Classic RISC 5-stage integer pipeline just barely has in-order completion
  - Load takes two cycles, but following one-cycle integer op completes at same time, not earlier
  - Adding pipelined FPU immediately brings OoO completion

In-Order versus Out-of-Order Commit

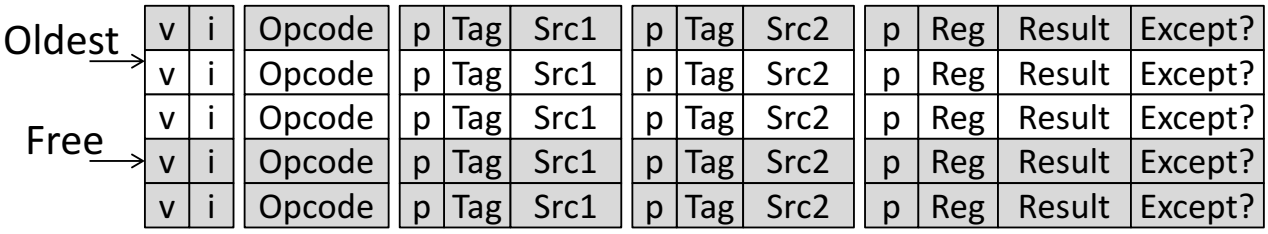
- In-order commit supports precise traps, standard today
  - Some proposals to reduce the cost of in-order commit by retiring some instructions early to compact reorder buffer, but this is just an optimized in-order commit
- Out-of-order commit was effectively what early OoO machines implemented (imprecise traps) as completion irrevocably changed machine state
  - i.e., complete == commit in these machines

OoO Design Choices

- Where are reservation stations?
  - Part of reorder buffer, or in separate issue window?
  - Distributed by functional units, or centralized?
- How is register renaming performed?
  - Tags and data held in reservation stations, with separate architectural register file
  - Tags only in reservation stations, data held in unified physical register file

"Data-in-ROB" Design

(HP PA8000, Pentium Pro, Core2Duo, Nehalem)



- Managed as circular buffer in program order, new instructions dispatched to free slots, oldest instruction committed/reclaimed when done ("p" bit set on result)
- Tag is given by index in ROB (Free pointer value)
- In dispatch, non-busy source operands read from architectural register file and copied to Src1 and Src2 with presence bit "p" set. Busy operands copy tag of producer and clear "p" bit.
- Set valid bit "v" on dispatch, set issued bit "i" on issue
- On completion, search source tags, set "p" bit and copy data into src on tag match. Write result and exception flags to ROB.
- On commit, check exception status, and copy result into architectural register file if no trap.
- On trap, flush machine and ROB, set free=oldest, jump to handler

Managing Rename for Data-in-ROB

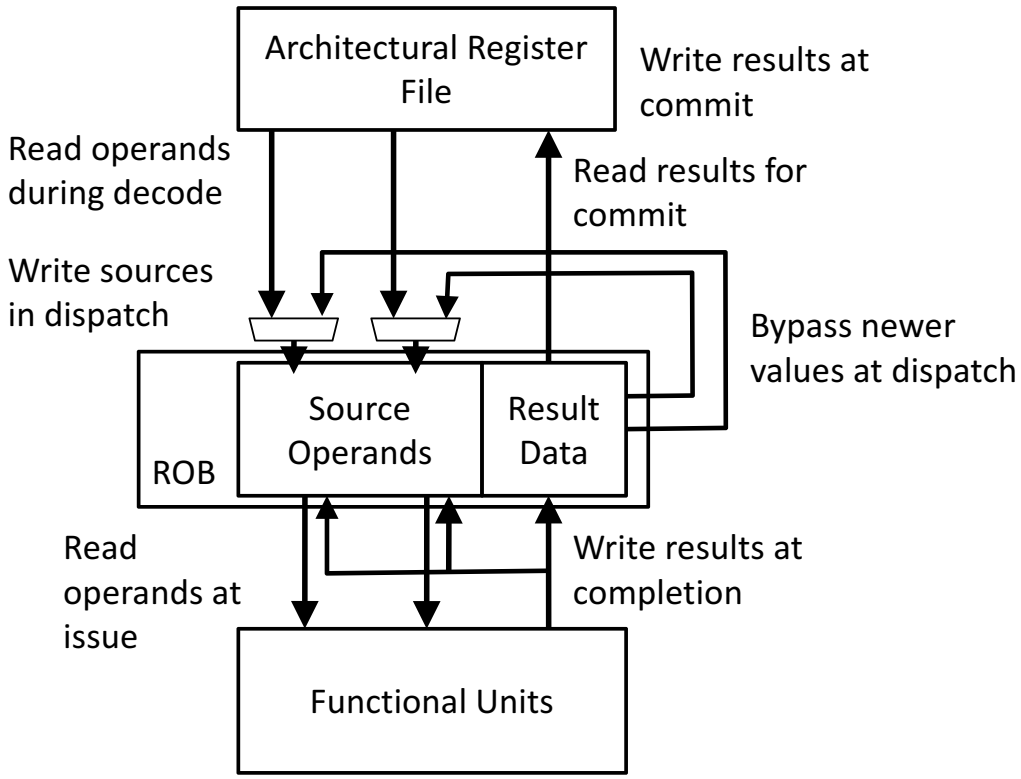
Rename table associated with architectural registers, managed in decode/dispatch

p	Tag	Value
p	Tag	Value
p	Tag	Value
p	Tag	Value

One entry per arch. register

- If "p" bit set, then use value in architectural register file
- Else, tag field indicates instruction that will/has produced value
- For dispatch, read source operands <p,tag,value> from arch. regfile, and also read <p,result> from producing instruction in ROB, bypassing as needed. Copy to ROB
- Write destination arch. register entry with <0,Free,_>, to assign tag to ROB index of this instruction
- On commit, update arch. regfile with <1, _, Result>
- On trap, reset table (All p=1)

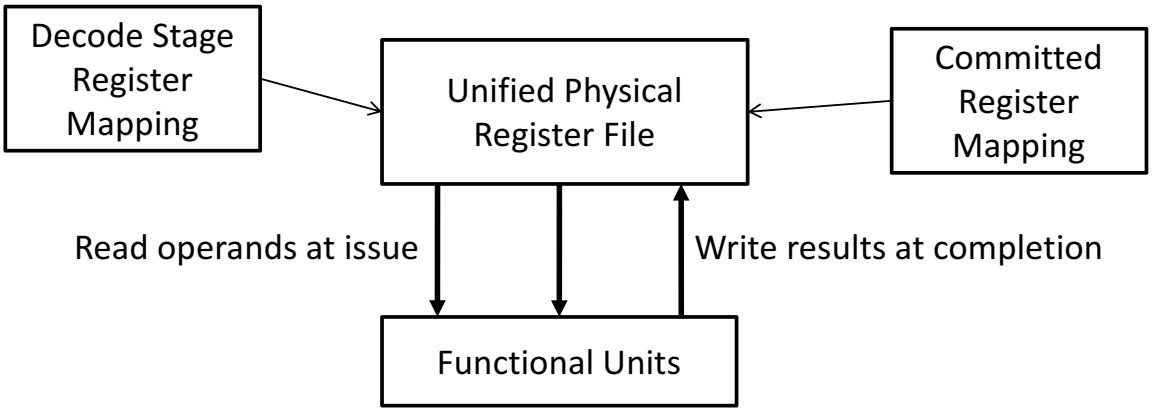
Data Movement in Data-in-ROB Design



Unified Physical Register File

(MIPS R10K, Alpha 21264, Intel Pentium 4 & Sandy/Ivy Bridge)

- Rename all architectural registers into a single *physical* register file during decode, no register values read
- Functional units read and write from single unified register file holding committed and temporary registers in execute
- Commit only updates mapping of architectural register to physical register, no data movement



Lifetime of Physical Registers

- Physical regfile holds committed and speculative values
- Physical registers decoupled from ROB entries *(no data in ROB)*

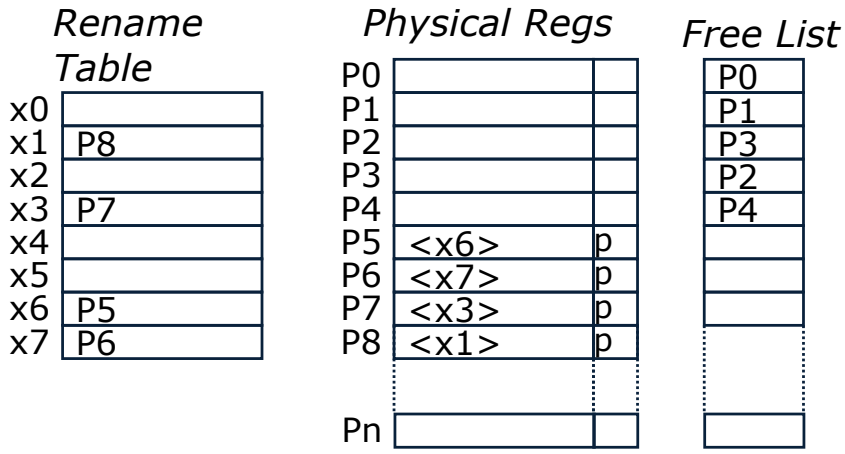
```
ld x1, (x3)              ld P1, (Px)
addi x3, x1, #4          addi P2, P1, #4
sub x6, x7, x9           sub P3, Py, Pz
add x3, x3, x6           add P4, P2, P3
ld x6, (x1)              ld P5, (P1)
add x6, x6, x3           add P6, P5, P4
sd x6, (x1)              sd P6, (P1)
ld x6, (x11)             ld P7, (Pw)
```

Rename

When can we reuse a physical register?

When next writer of same architectural register commits

Physical Register Management



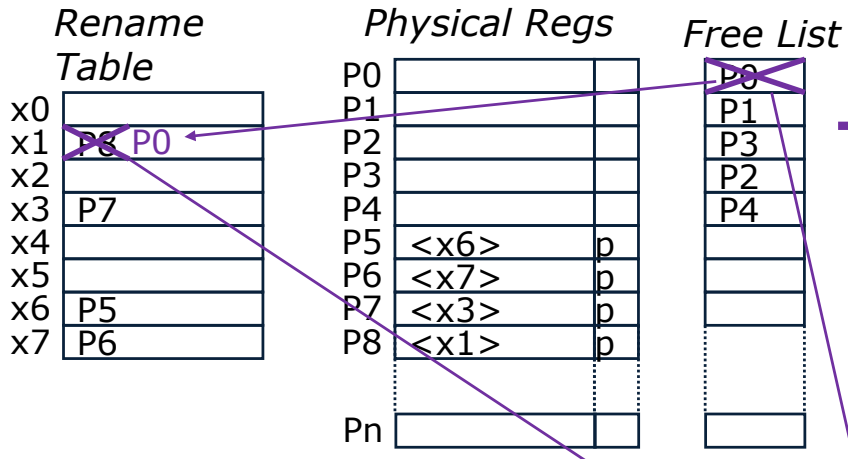

```
ld x1, 0(x3)
addi x3, x1, #4
sub x6, x7, x6
add x3, x3, x6
ld x6, 0(x1)
```

ROB

use	ex	op	p1	PR1	p2	PR2	Rd	LPRd	PRd

(LPRd requires third read port on Rename Table for each instruction)

Physical Register Management

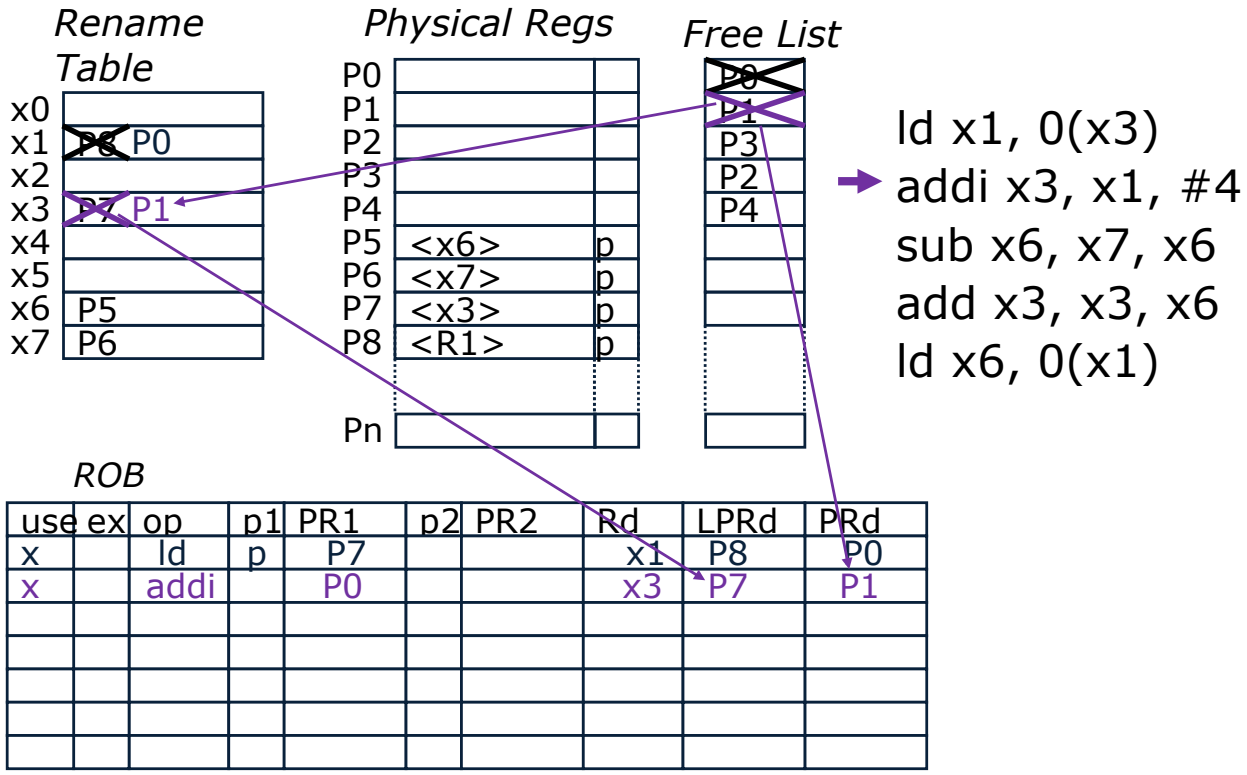



```
→ ld x1, 0(x3)
addi x3, x1, #4
sub x6, x7, x6
add x3, x3, x6
ld x6, 0(x1)
```

ROB

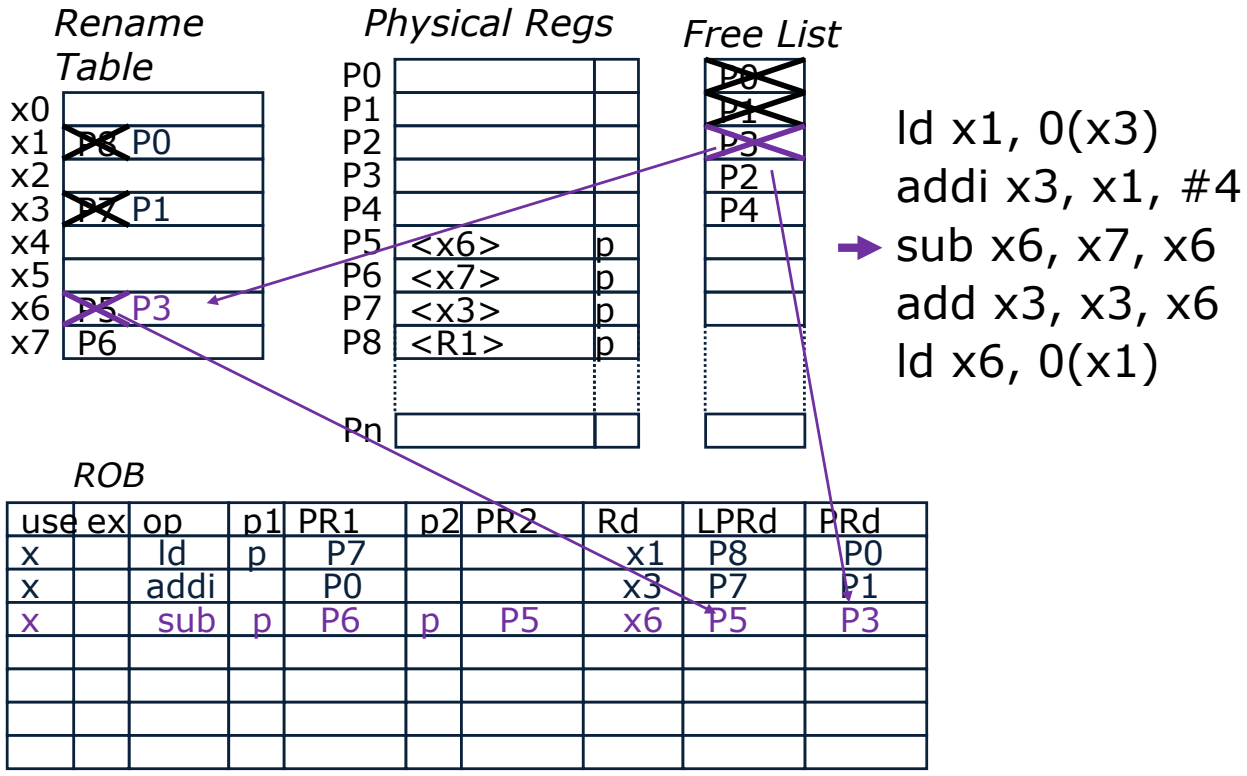
use	ex	op	p1	PR1	p2	PR2	Rd	LPRd	PRd
x		ld	p	P7			x1	P8	P0

Physical Register Management



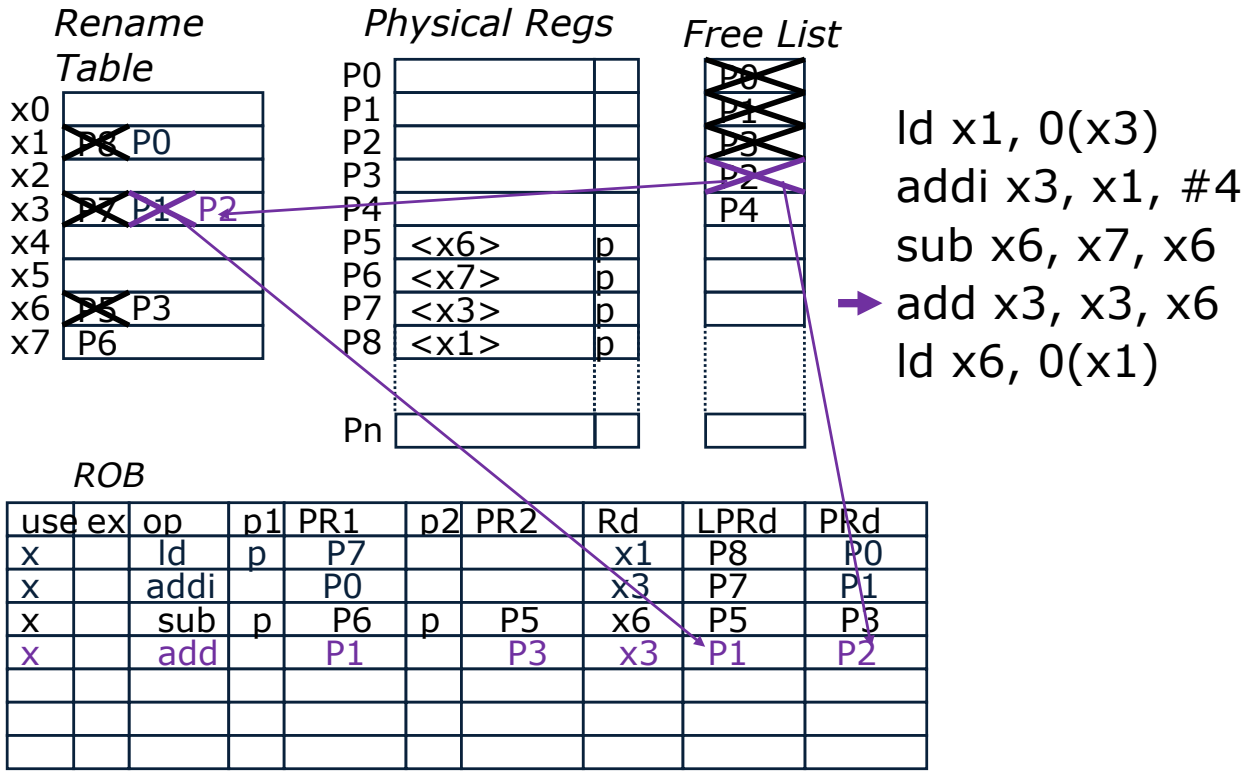
use	ex	op	p1	PR1	p2	PR2	Rd	LPRd	PRd
x		ld	p	P7			x1	P8	P0
x		addi		P0			x3	P7	P1

Physical Register Management

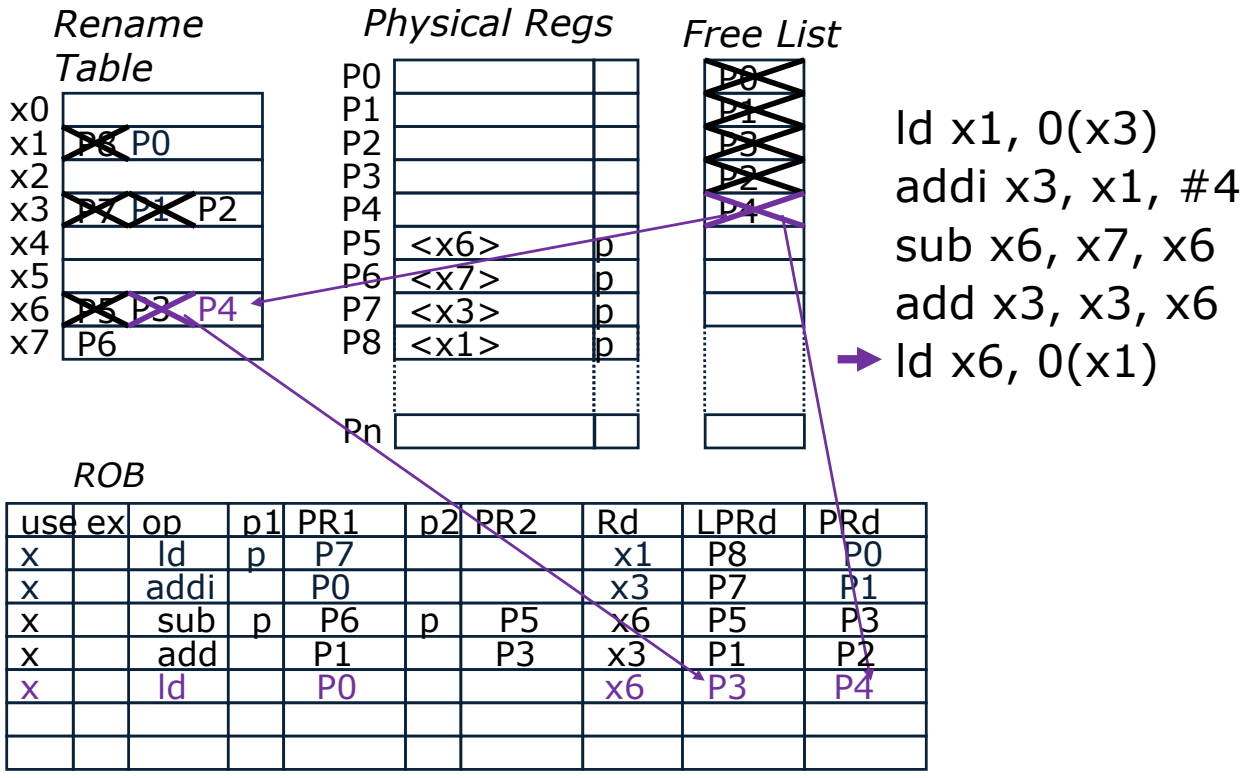


use	ex	op	p1	PR1	p2	PR2	Rd	LPRd	PRd
x		ld	p	P7			x1	P8	P0
x		addi		P0			x3	P7	P1
x		sub	p	P6	p	P5	x6	P5	P3

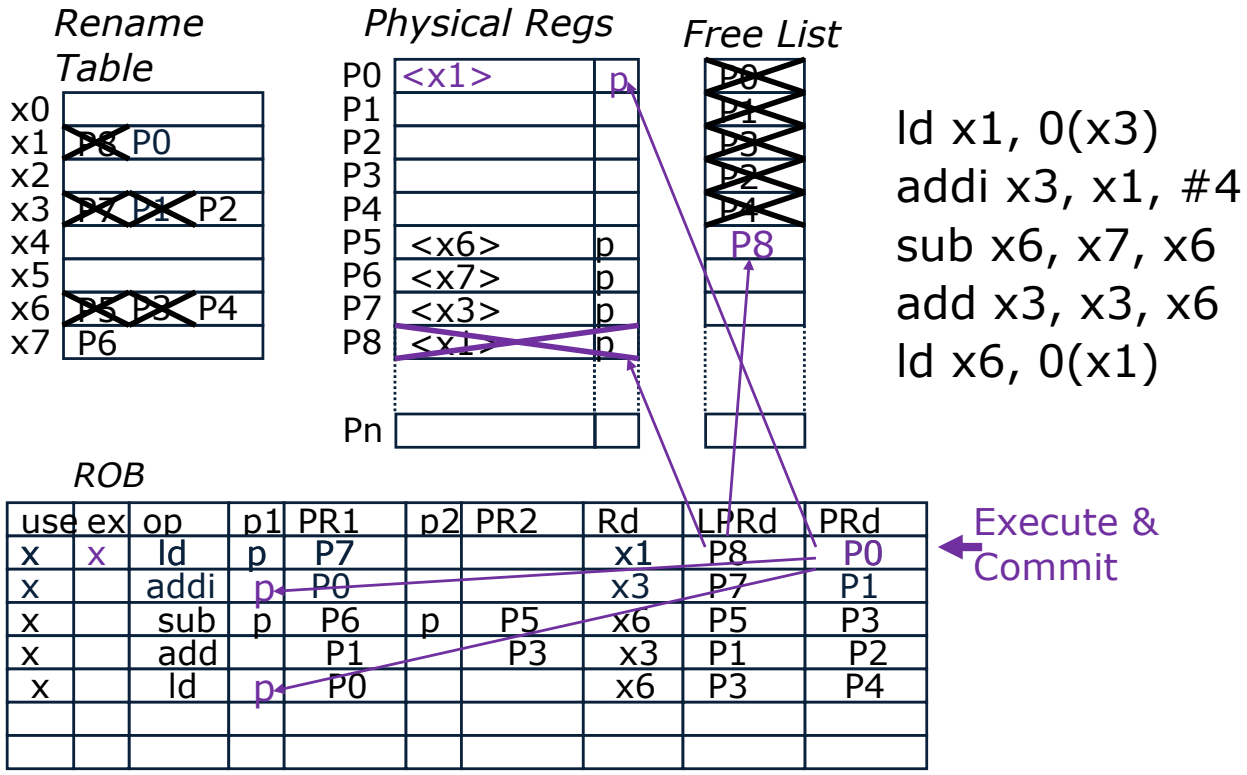
Physical Register Management



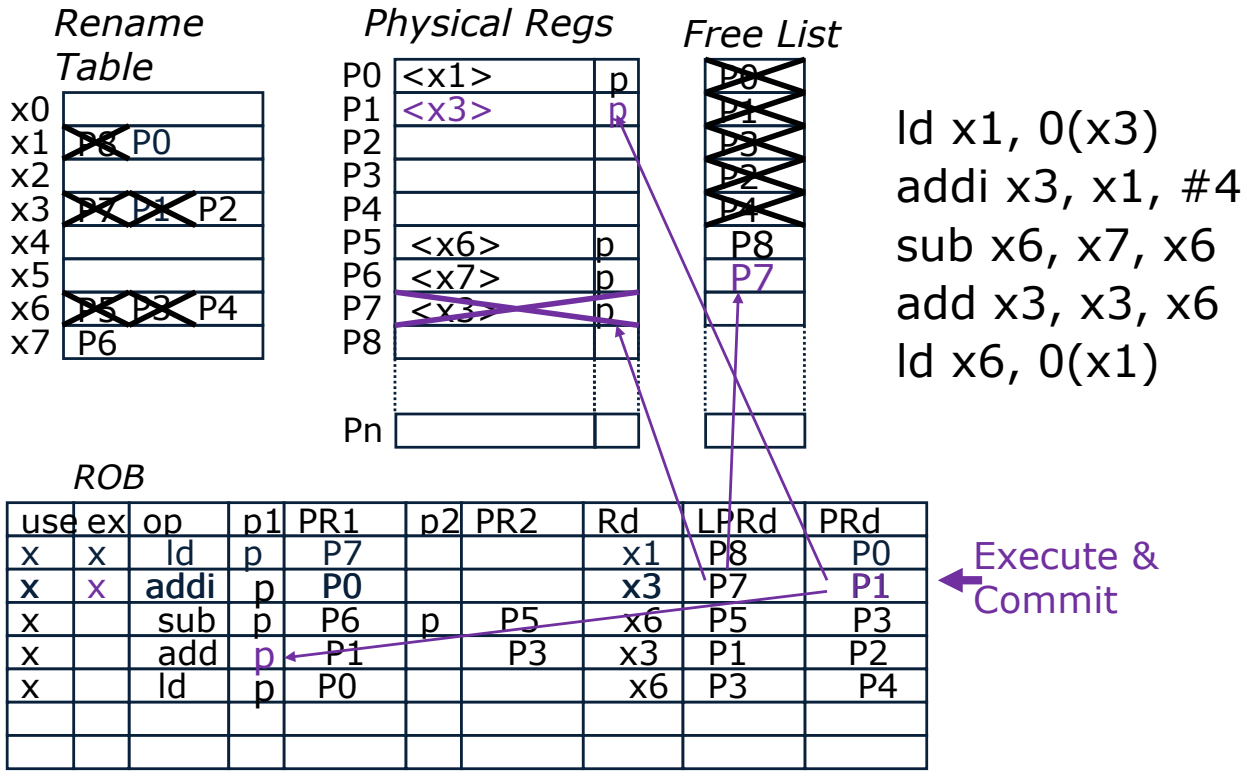
Physical Register Management



Physical Register Management



Physical Register Management

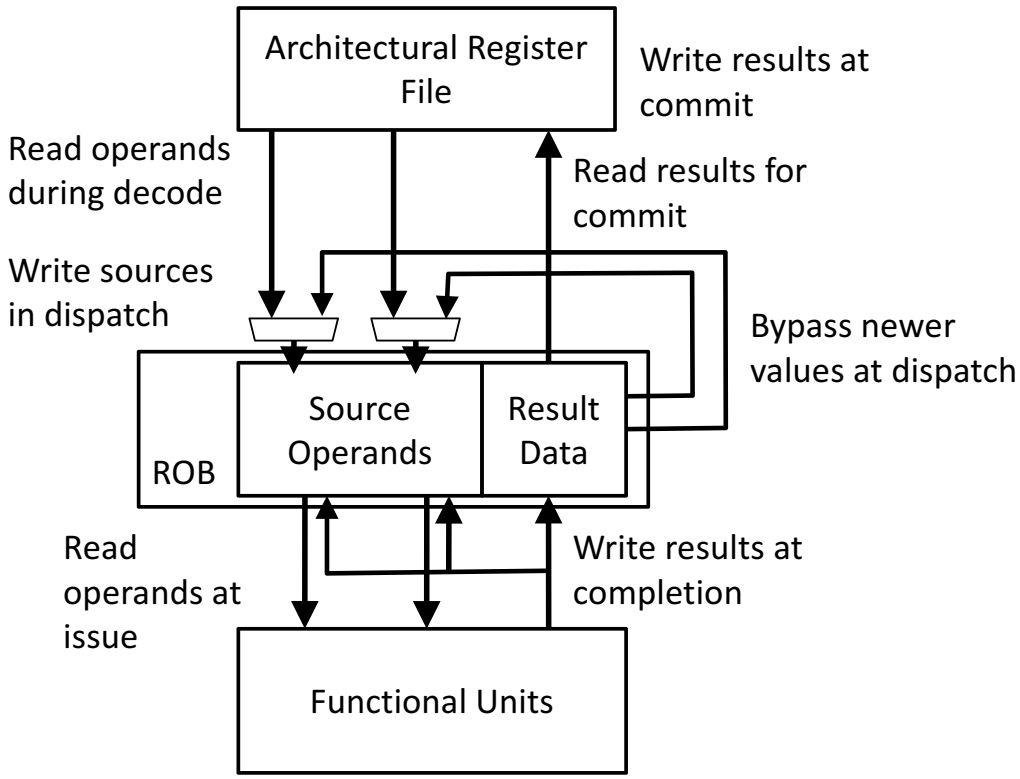


MIPS R10K Trap Handling

- Rename table is repaired by unrenaming instructions in reverse order using the PRd/LPRd fields
- The Alpha 21264 had similar physical register file scheme, but kept complete rename table snapshots for each instruction in ROB (80 snapshots total)
  - Flash copy all bits from snapshot to active table in one cycle

Part II: Advanced Out-of-Order Superscalar Designs

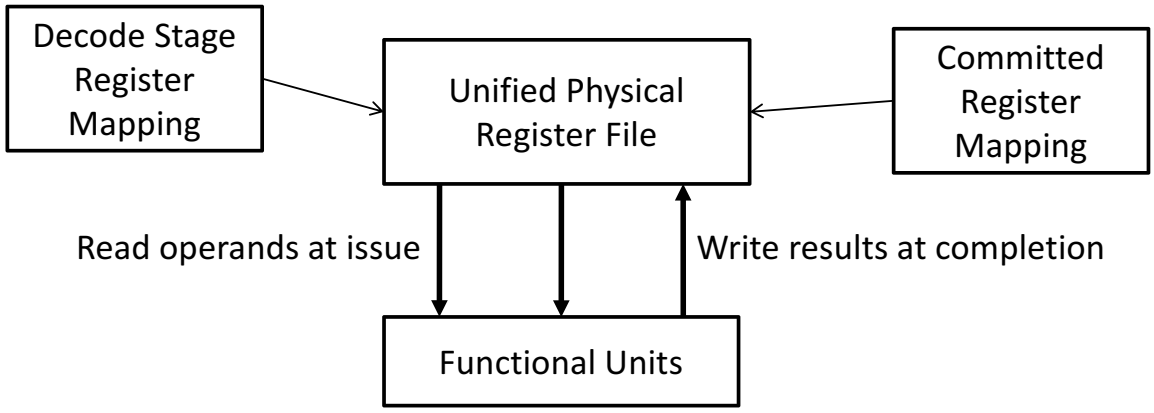
Data Movement in Data-in-ROB Design



Unified Physical Register File

(MIPS R10K, Alpha 21264, Intel Pentium 4 & Sandy/Ivy Bridge)

- Rename all architectural registers into a single *physical* register file during decode, no register values read
- Functional units read and write from single unified register file holding committed and temporary registers in execute
- Commit only updates mapping of architectural register to physical register, no data movement



Lifetime of Physical Registers

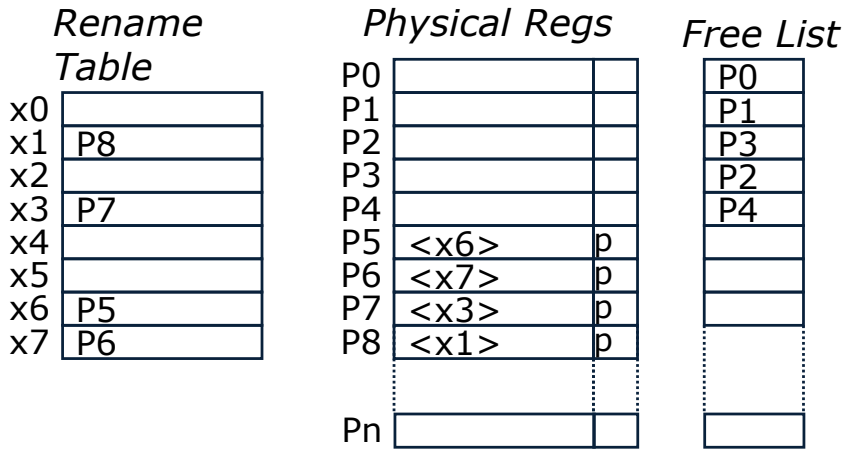
- Physical regfile holds committed and speculative values
- Physical registers decoupled from ROB entries *(no data in ROB)*

```
ld x1, (x3)                ld P1, (Px)
addi x3, x1, #4            addi P2, P1, #4
sub x6, x7, x9             sub P3, Py, Pz
add x3, x3, x6             add P4, P2, P3
ld x6, (x1)      Rename    ld P5, (P1)
add x6, x6, x3             add P6, P5, P4
sd x6, (x1)                sd P6, (P1)
ld x6, (x11)               ld P7, (Pw)
```

When can we reuse a physical register?

When next writer of same architectural register commits

Physical Register Management



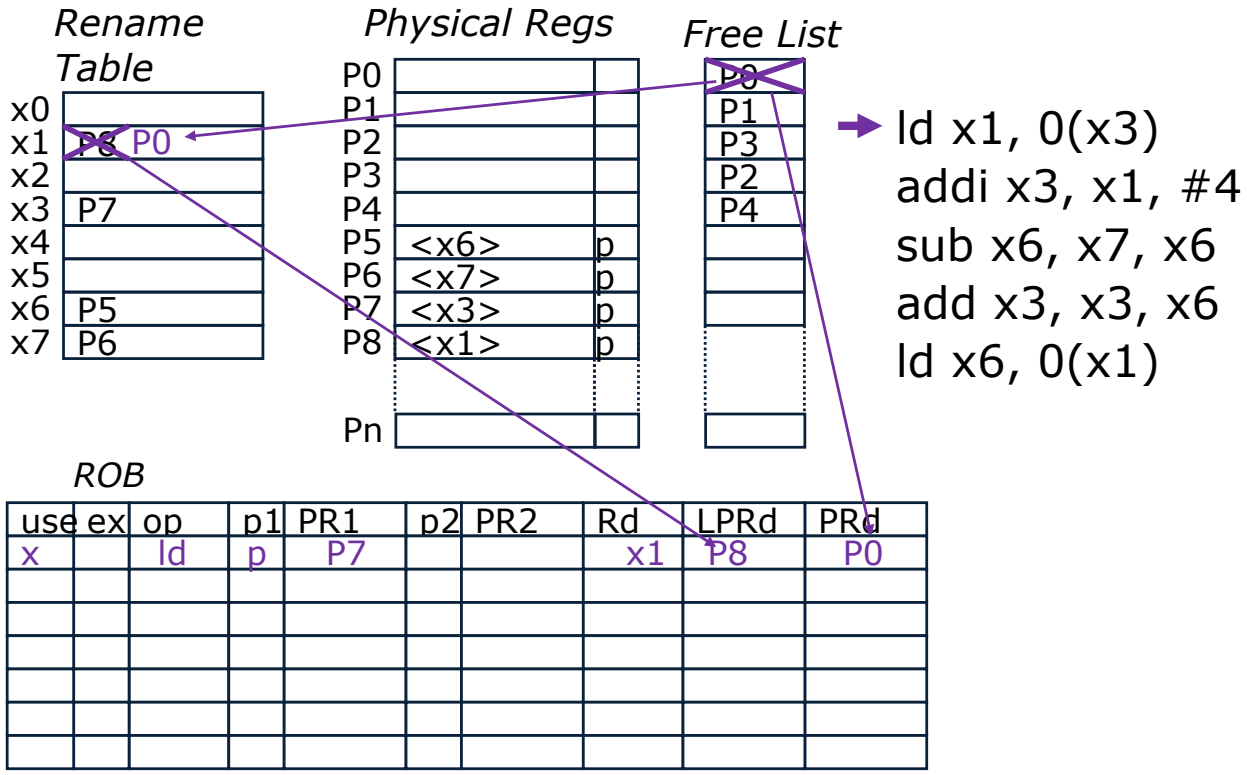
ld x1, 0(x3)
addi x3, x1, #4
sub x6, x7, x6
add x3, x3, x6
ld x6, 0(x1)

ROB

use	ex	op	p1	PR1	p2	PR2	Rd	LPRd	PRd

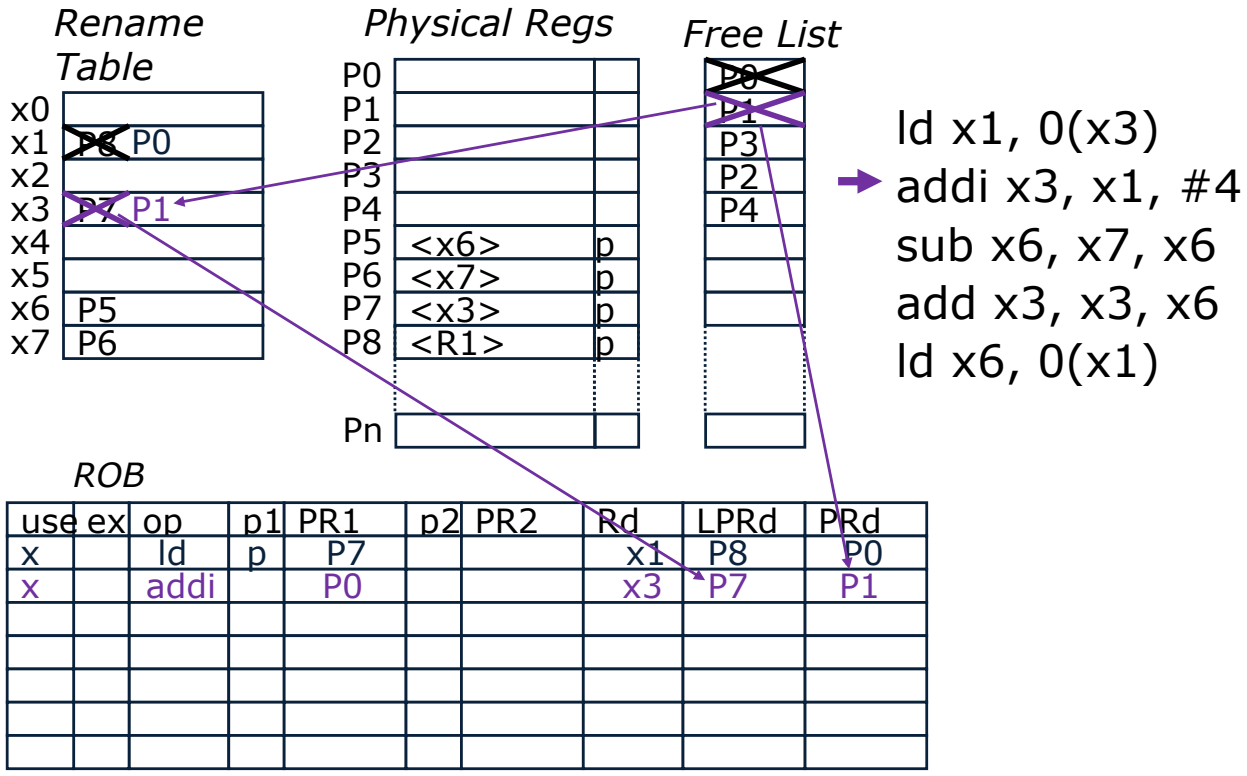
(LPRd requires third read port on Rename Table for each instruction)

Physical Register Management



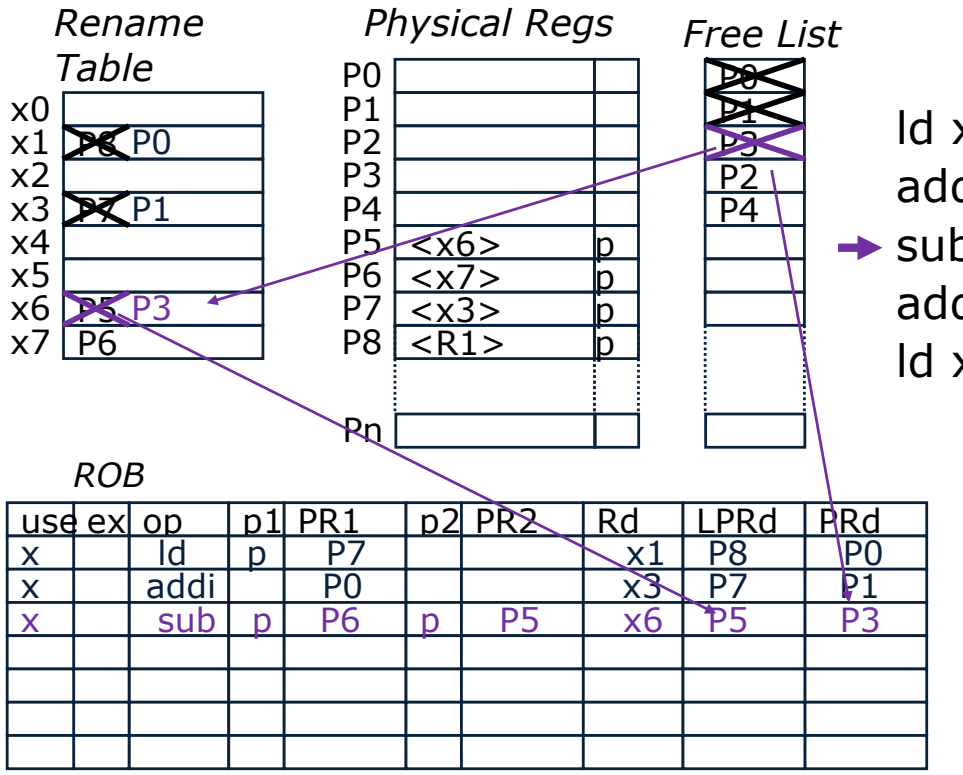
use	ex	op	p1	PR1	p2	PR2	Rd	LPRd	PRd
x		ld	p	P7			x1	P8	P0

Physical Register Management



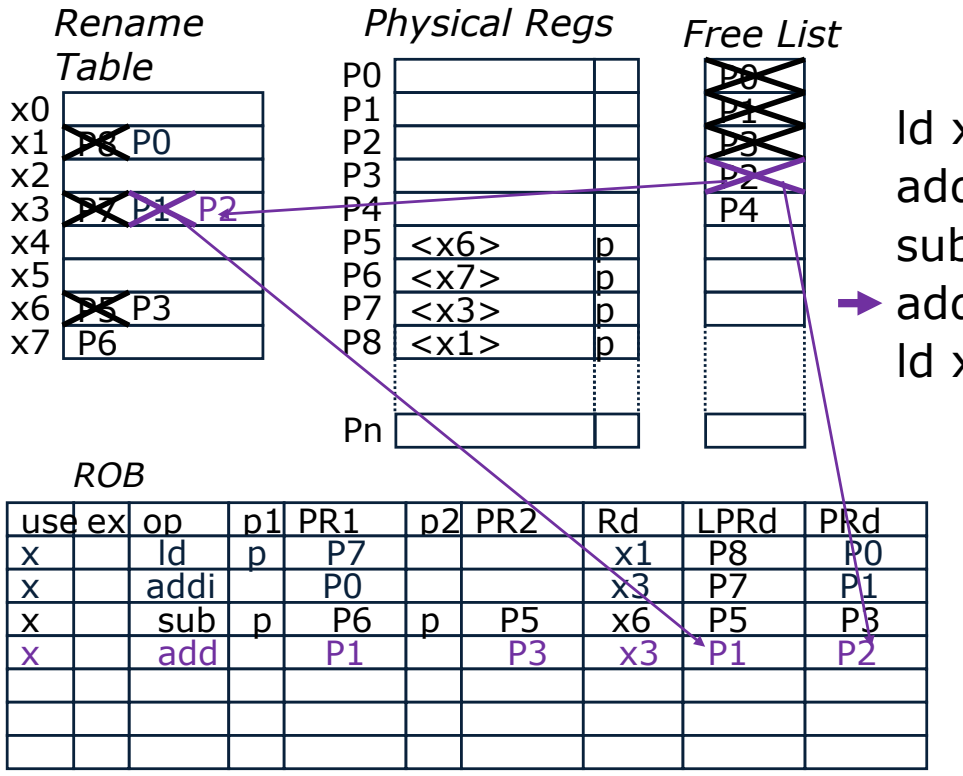
use	ex	op	p1	PR1	p2	PR2	Rd	LPRd	PRd
x		ld	p	P7			x1	P8	P0
x		addi		P0			x3	P7	P1

Physical Register Management



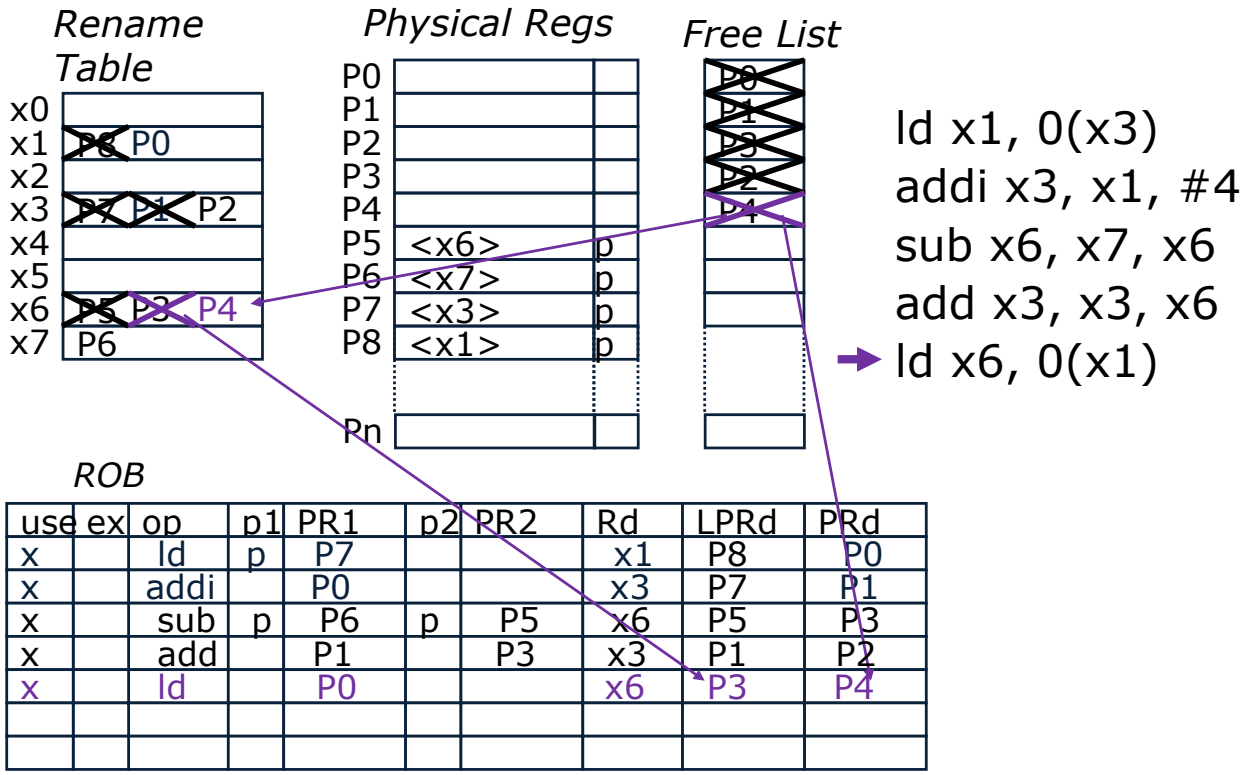
use	ex	op	p1	PR1	p2	PR2	Rd	LPRd	PRd
x		ld	p	P7			x1	P8	P0
x		addi		P0			x3	P7	P1
x		sub	p	P6	p	P5	x6	P5	P3

Physical Register Management



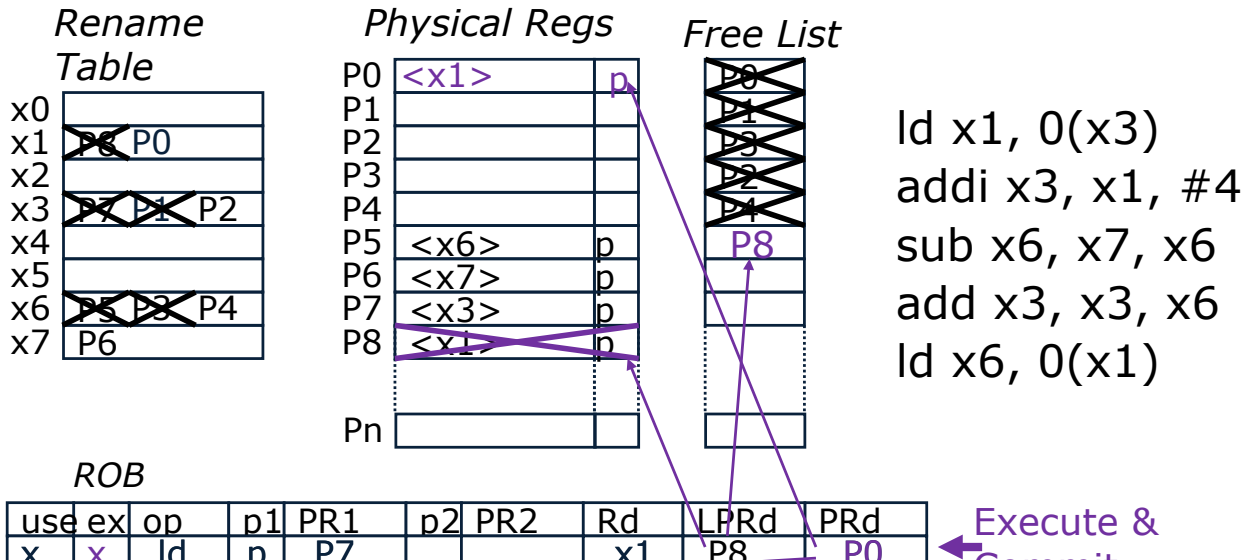
use	ex	op	p1	PR1	p2	PR2	Rd	LPRd	PRd
x		ld	p	P7			x1	P8	P0
x		addi		P0			x3	P7	P1
x		sub	p	P6	p	P5	x6	P5	P3
x		add		P1		P3	x3	P1	P2

Physical Register Management



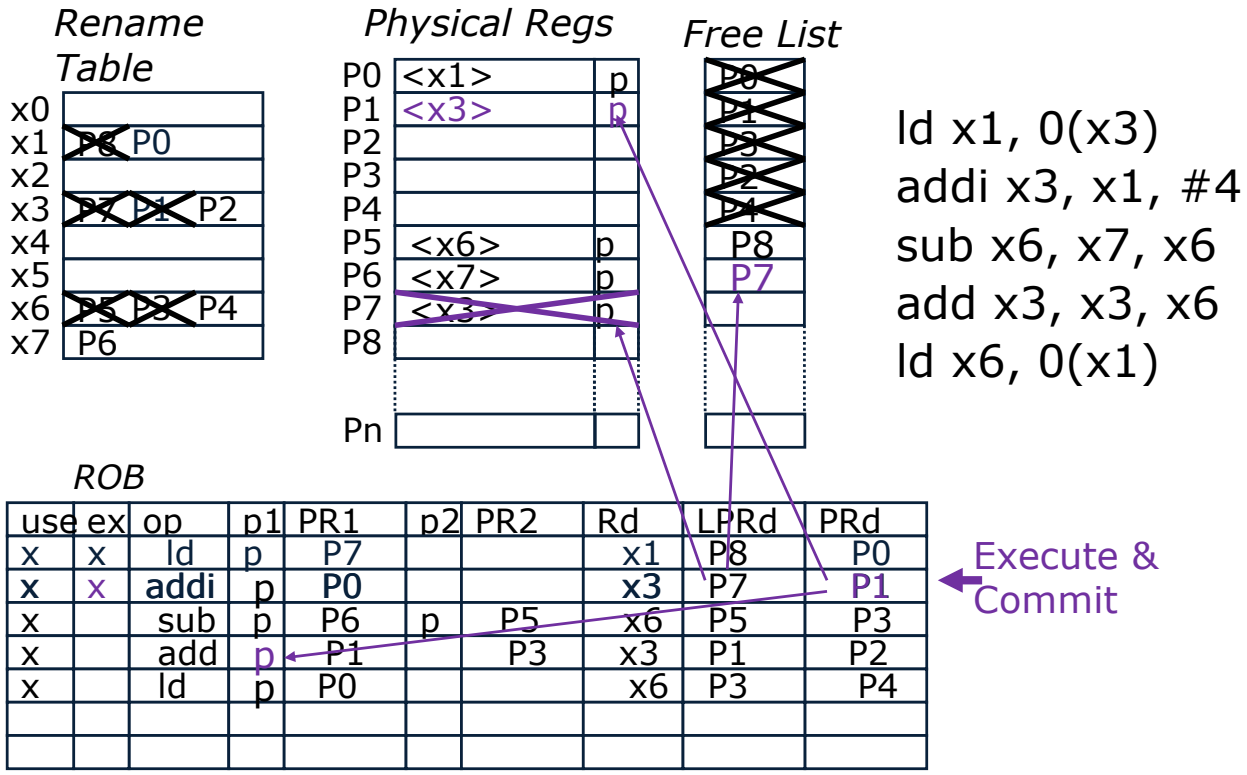
use	ex	op	p1	PR1	p2	PR2	Rd	LPRd	PRd
x		ld	p	P7			x1	P8	P0
x		addi		P0			x3	P7	P1
x		sub	p	P6	p	P5	x6	P5	P3
x		add		P1		P3	x3	P1	P2
x		ld		P0			x6	P3	P4

Physical Register Management



use	ex	op	p1	PR1	p2	PR2	Rd	LPRd	PRd
x	x	ld	p	P7			x1	P8	P0
x		addi	p	P0			x3	P7	P1
x		sub	p	P6	p	P5	x6	P5	P3
x		add		P1		P3	x3	P1	P2
x		ld	p	P0			x6	P3	P4

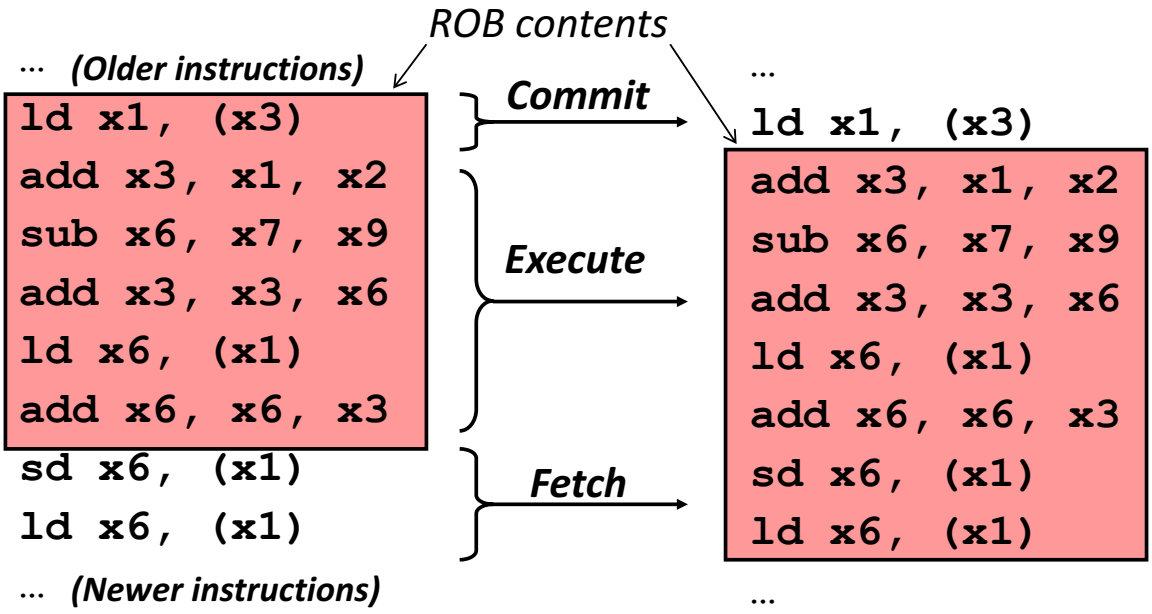
Physical Register Management



MIPS R10K Trap Handling

- Rename table is repaired by unrenaming instructions in reverse order using the PRd/LPRd fields
- The Alpha 21264 had similar physical register file scheme, but kept complete rename table snapshots for each instruction in ROB (80 snapshots total)
  - Flash copy all bits from snapshot to active table in one cycle

Reorder Buffer Holds Active Instructions (Decoded but not Committed)



Separate Issue Window from ROB

The issue window holds only instructions that have been decoded and renamed but not issued into execution. Has register tags and presence bits, and pointer to ROB entr

use	ex	op	p1	PR1	p2	PR2	PRd	ROB#

Reorder buffer used to hold exception information for commit.

Oldest

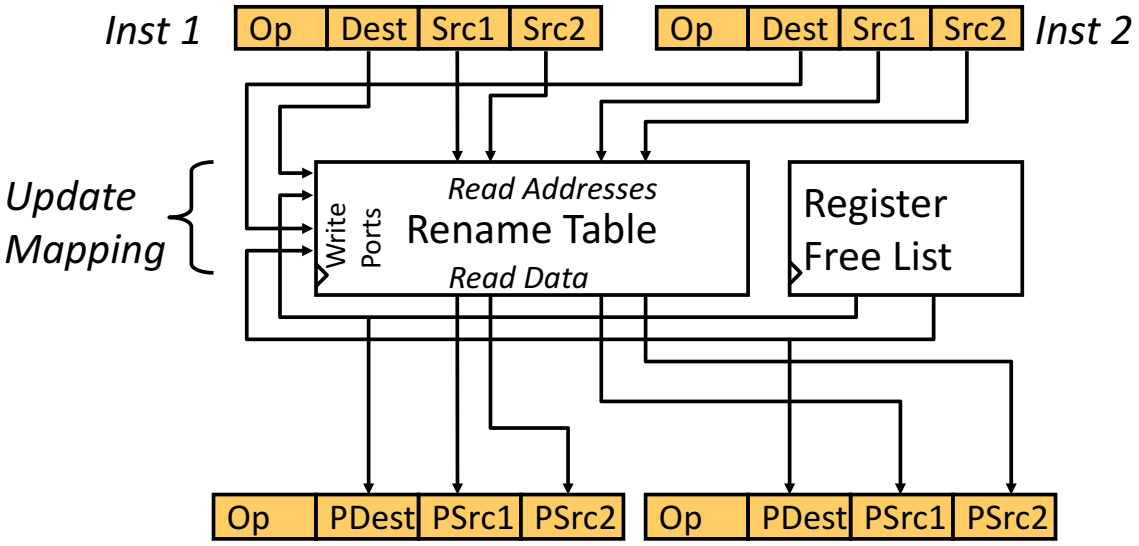
Free

Done?	Rd	LPRd	PC	Except?

ROB is usually several times larger than issue window – why?

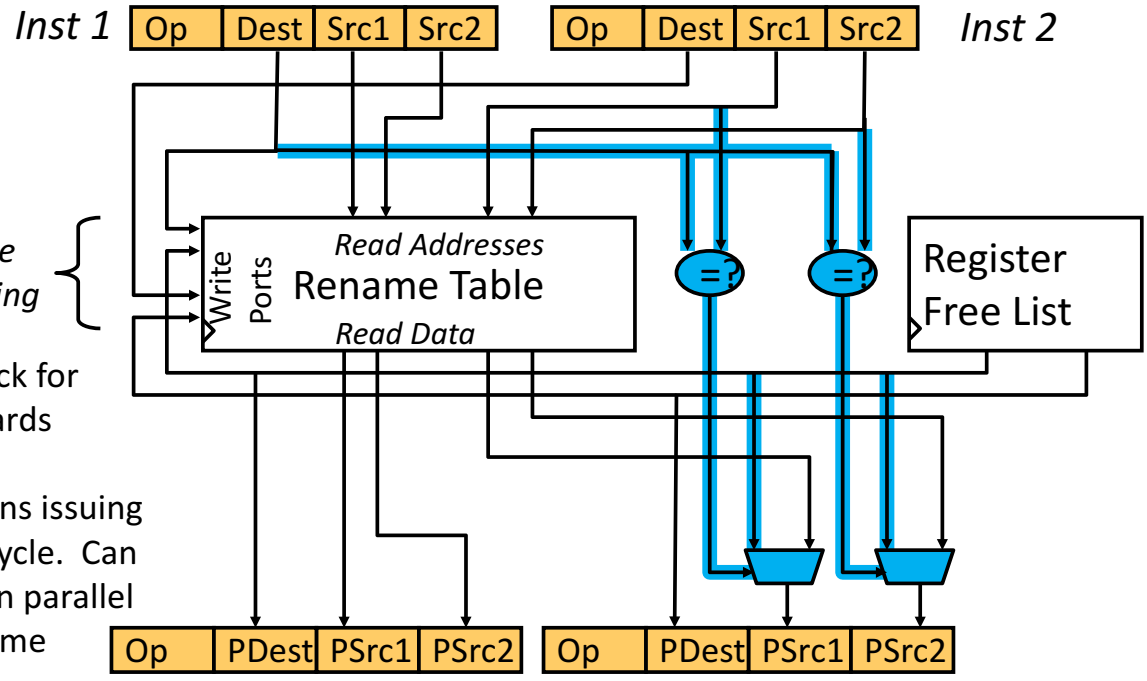
Superscalar Register Renaming

- During decode, instructions allocated new physical destination register
- Source operands renamed to physical register with newest value
- Execution unit only sees physical register numbers



Does this work?

Superscalar Register Renaming



Must check for RAW hazards between instructions issuing in same cycle. Can be done in parallel with rename lookup.

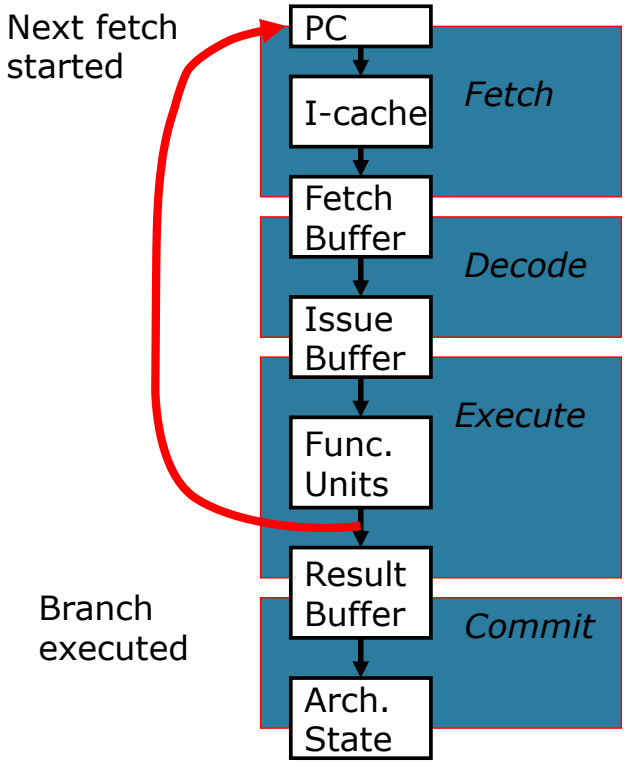
MIPS R10K renames 4 serially-RAW-dependent insts/cycle

Control Flow Penalty

Modern processors may have > 10 pipeline stages between next PC calculation and branch resolution !

How much work is lost if pipeline doesn't follow correct instruction flow?

~ Loop length x pipeline width + buffers



Reducing Control Flow Penalty

- Software solutions
  - Eliminate branches - loop unrolling
    - Increases the run length
  - Reduce resolution time - instruction scheduling
    - Compute the branch condition as early as possible (of limited value because branches often in critical path through code)

- Hardware solutions
  - Find something else to do - delay slots
    - Replaces pipeline bubbles with useful work (requires software cooperation) – quickly see diminishing returns
  - Speculate - branch prediction
    - Speculative execution of instructions beyond the branch
    - Many advances in accuracy

Branch Prediction

Motivation:

Branch penalties limit performance of deeply pipelined processors

Modern branch predictors have high accuracy (>95%) and can reduce branch penalties significantly

Required hardware support:

Prediction structures:

- Branch history tables, branch target buffers, etc.

Mispredict recovery mechanisms:

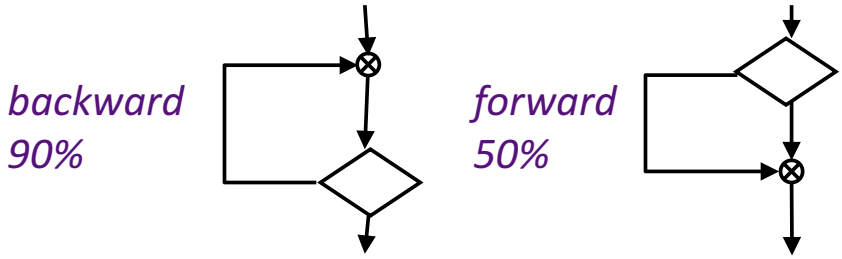
- *Keep result computation separate from commit*
- Kill instructions following branch in pipeline
- Restore state to that following branch

Importance of Branch Prediction

- Consider 4-way superscalar with 8 pipeline stages from fetch to dispatch, and 80-entry ROB, and 3 cycles from issue to branch resolution
- On a mispredict, could throw away 8*4+(80-1)=111 instructions
- Improving from 90% to 95% prediction accuracy, removes 50% of branch mispredicts
  - If 1/6 instructions are branches, then move from 60 instructions between mispredicts, to 120 instructions between mispredicts

Static Branch Prediction

Overall probability a branch is taken is ~60-70% but:



ISA can attach preferred direction semantics to branches, e.g., Motorola MC88110

bne0 *(preferred taken)* beq0 *(not taken)*

ISA can allow arbitrary choice of statically predicted direction, e.g., HP PA-RISC, Intel IA-64

typically reported as ~80% accurate

Dynamic Branch Prediction learning based on past behavior

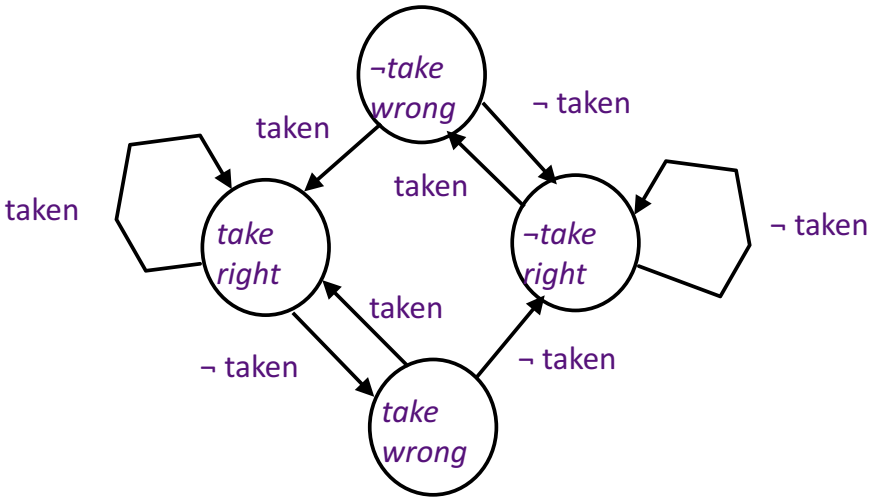
- Temporal correlation
  - The way a branch resolves may be a good predictor of the way it will resolve at the next execution
- Spatial correlation
  - Several branches may resolve in a highly correlated manner (a preferred path of execution)

One-Bit Branch History Predictor

- For each branch, remember last way branch went
- Has problem with loop-closing backward branches, as two mispredicts occur on every loop execution
  1. first iteration predicts loop backwards branch not-taken (loop was exited last time)
  2. last iteration predicts loop backwards branch taken (loop continued last time)

Branch Prediction Bits

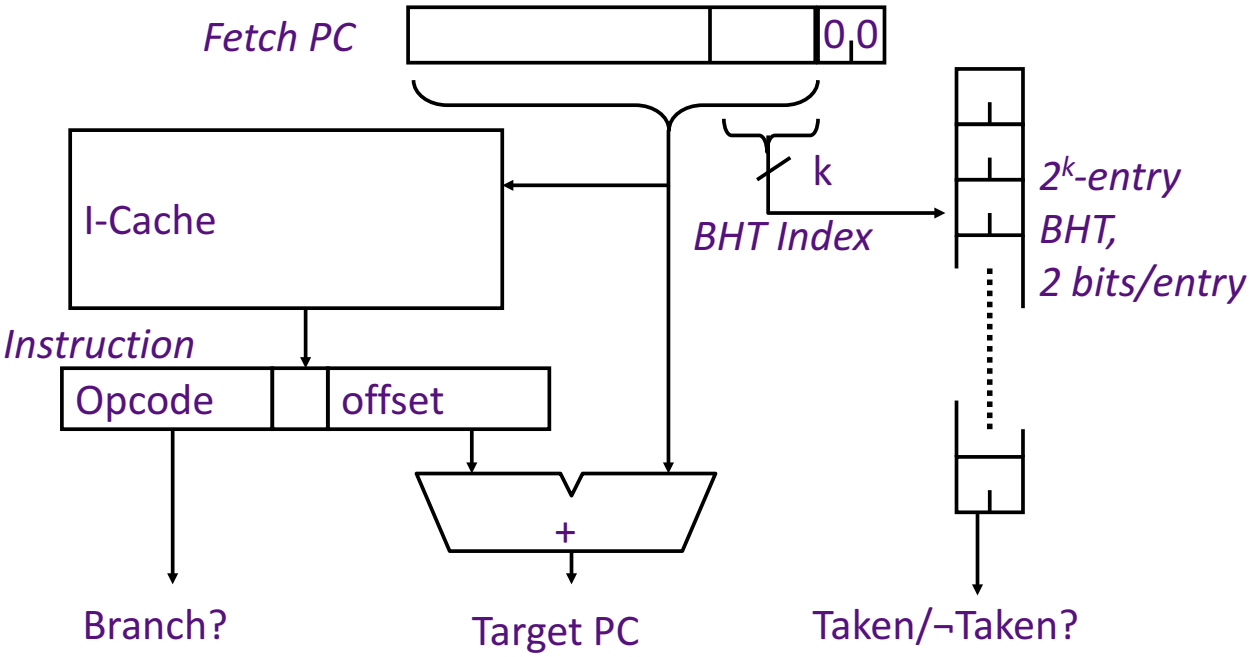
- Assume 2 BP bits per instruction
- Change the prediction after two consecutive mistakes!



BP state:

(*predict* take/¬take) x (*last prediction* right/wrong)

Branch History Table (BHT)



4K-entry BHT, 2 bits/entry, ~80-90% correct predictions

Exploiting Spatial Correlation

Yeh and Patt, 1992

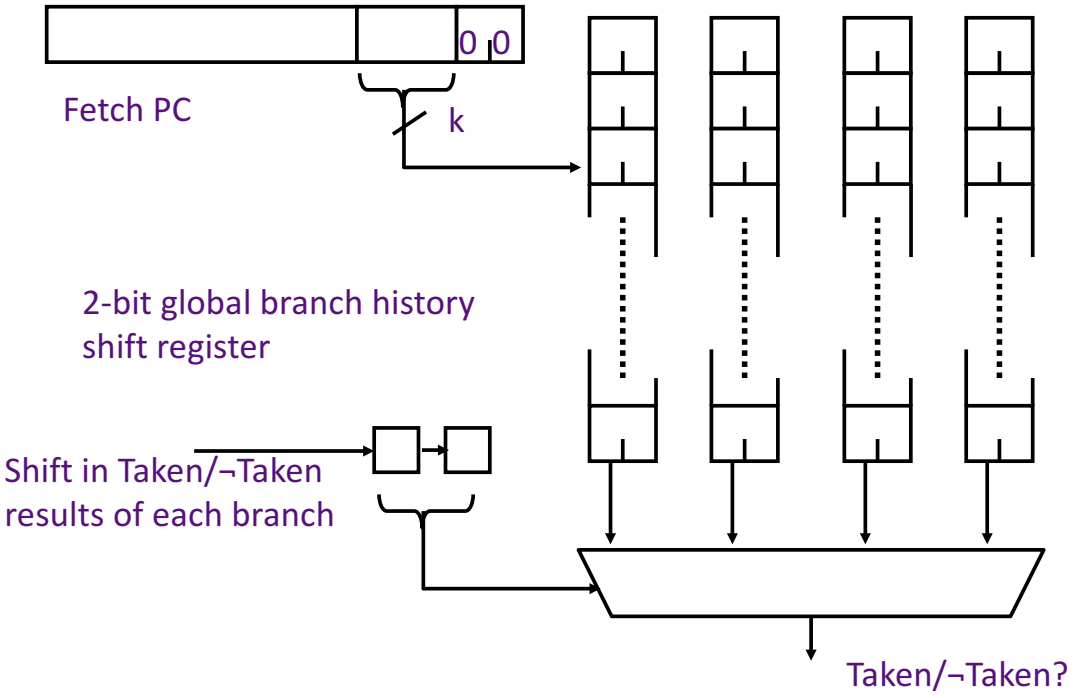
```
if (x[i] < 7) then
    y += 1;
if (x[i] < 5) then
    c -= 4;
```

If first condition false, second condition also false

History register, H, records the direction of the last N branches executed by the processor

Two-Level Branch Predictor

Pentium Pro uses the result from the last two branches to select one of the four sets of BHT bits (~95% correct)

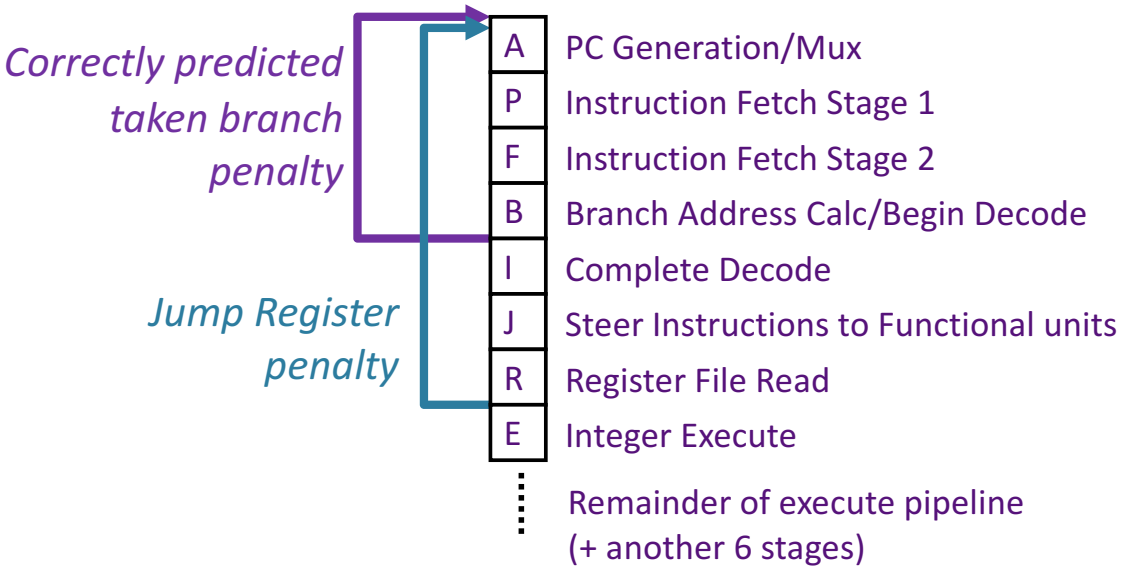


Speculating Both Directions

- An alternative to branch prediction is to execute both directions of a branch speculatively
  - resource requirement is proportional to the number of concurrent speculative executions
  - only half the resources engage in useful work when both directions of a branch are executed speculatively
  - branch prediction takes less resources than speculative execution of both paths
- With accurate branch prediction, it is more cost effective to dedicate all resources to the predicted direction!

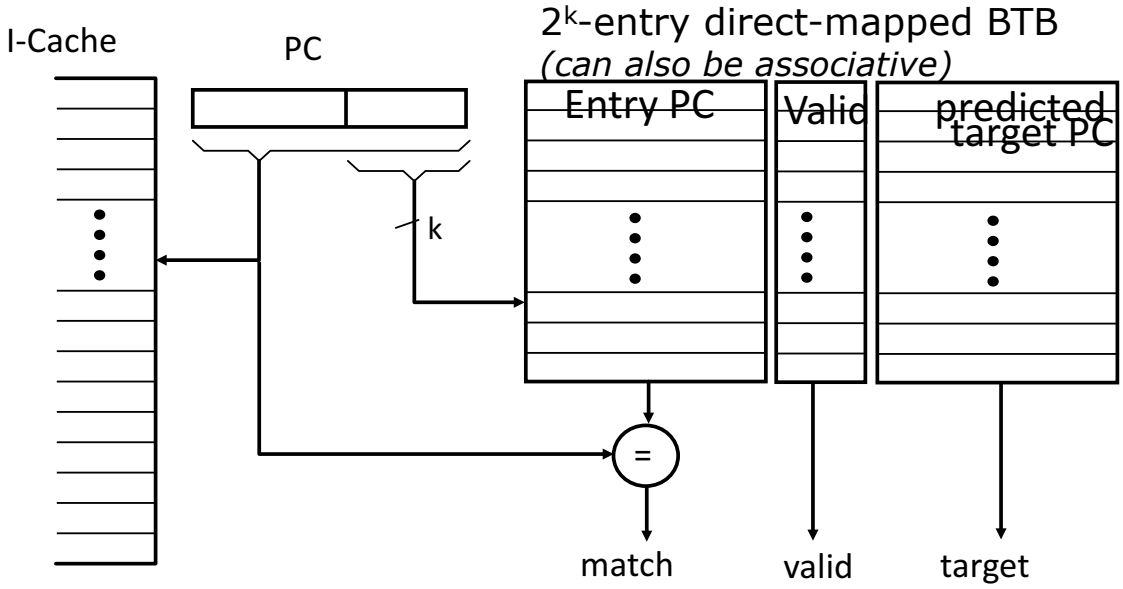
Limitations of BHTs

Only predicts branch direction. Therefore, cannot redirect fetch stream until after branch target is determined.



UltraSPARC-III fetch pipeline

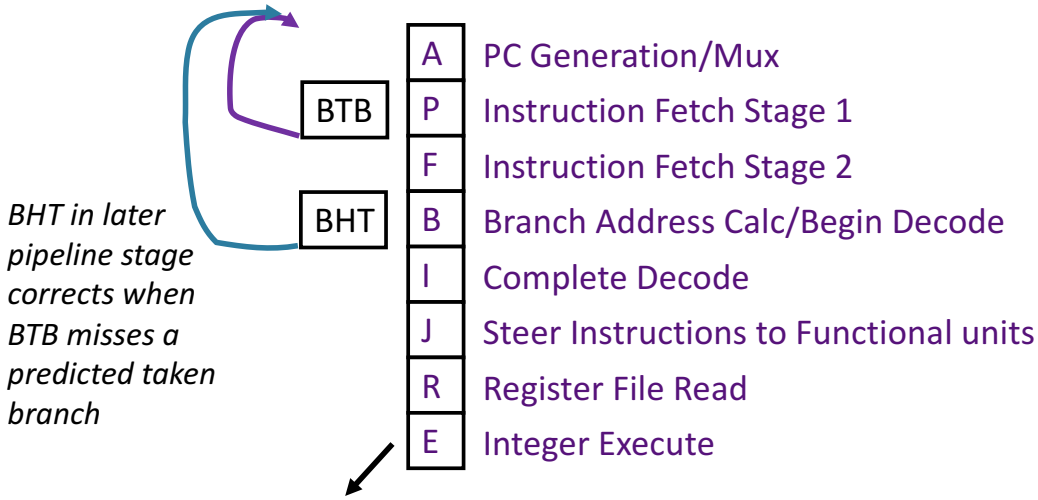
Branch Target Buffer (BTB)



- Keep both the branch PC and target PC in the BTB
- PC+4 is fetched if match fails
- Only *taken* branches and jumps held in BTB
- Next PC determined *before* branch fetched and decoded

Combining BTB and BHT

- BTB entries are considerably more expensive than BHT, but can redirect fetches at earlier stage in pipeline and can accelerate indirect branches (JR)
- BHT can hold many more entries and is more accurate



BTB/BHT only updated after branch resolves in E stage

Uses of Jump Register (JR)

- Switch statements (jump to address of matching case)

  BTB works well if same case used repeatedly

- Dynamic function call (jump to run-time function address)

  BTB works well if same function usually called, (e.g., in C++ programming, when objects have same type in virtual function call)

- Subroutine returns (jump to return address)

  BTB works well if usually return to the same place

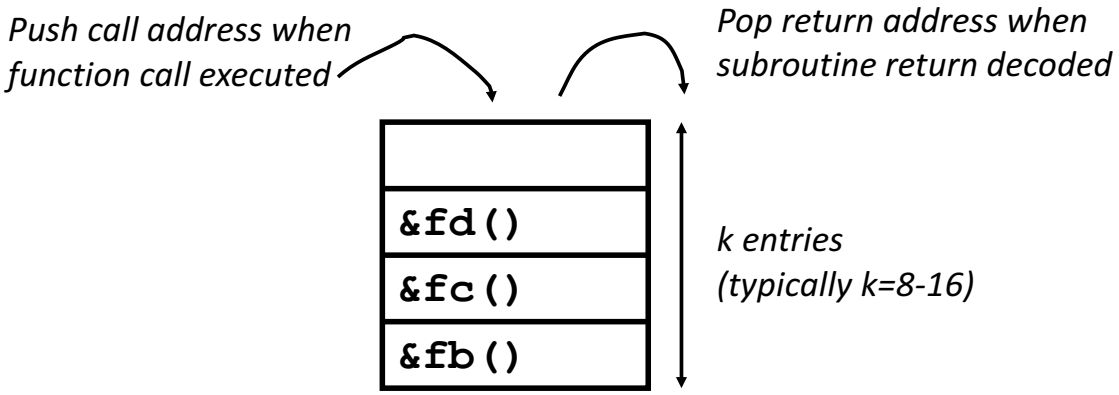
  ⇒ *Often one function called from many distinct call sites!*

How well does BTB work for each of these cases?

Subroutine Return Stack

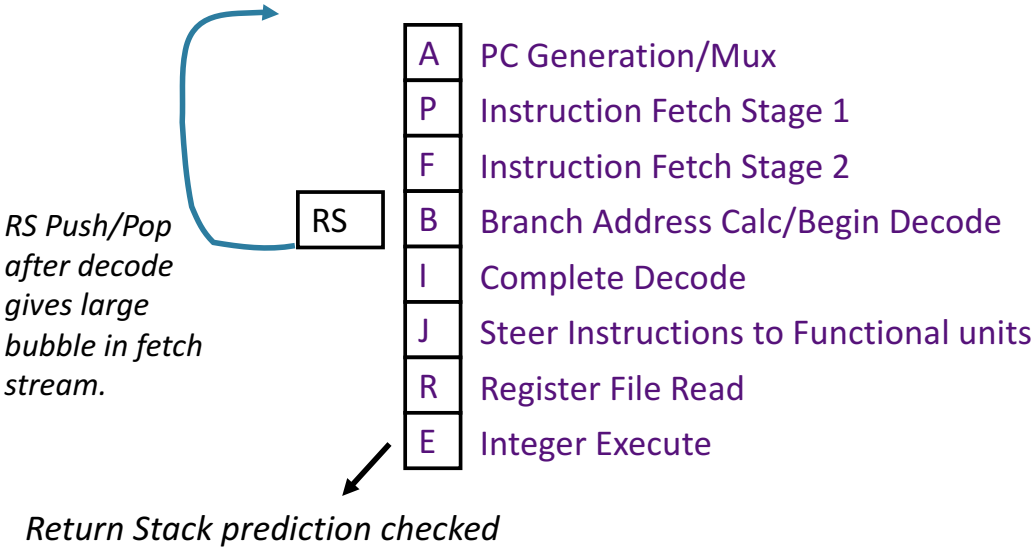
Small structure to accelerate JR for subroutine returns, typically much more accurate than BTBs.

```
fa() { fb(); }
fb() { fc(); }
fc() { fd(); }
```



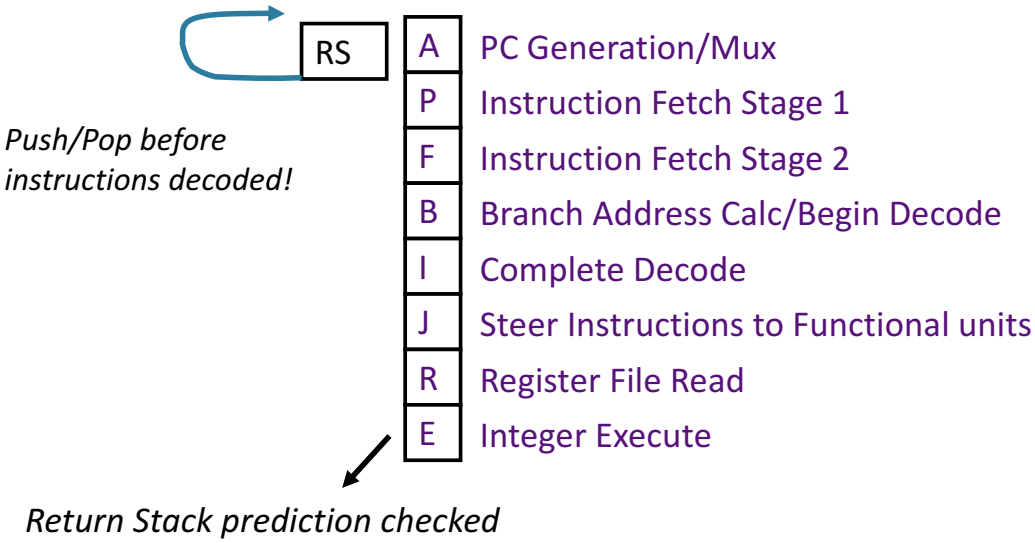
Return Stack in Pipeline

- How to use return stack (RS) in deep fetch pipeline?
- Only know if subroutine call/return at decode

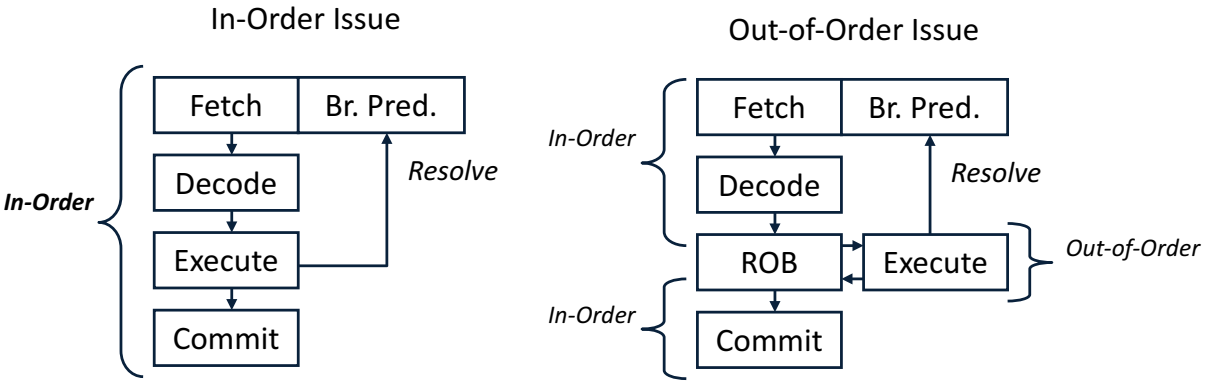


Return Stack in Pipeline

- Can remember whether PC is subroutine call/return using BTB-like structure
- Instead of target-PC, just store push/pop bit



In-Order vs. Out-of-Order Branch Prediction



- Speculative fetch but not speculative execution - branch resolves before later instructions complete
- Completed values held in bypass network until commit

- Speculative execution, with branches resolved after later instructions complete
- Completed values held in rename registers in ROB or unified physical register file until commit

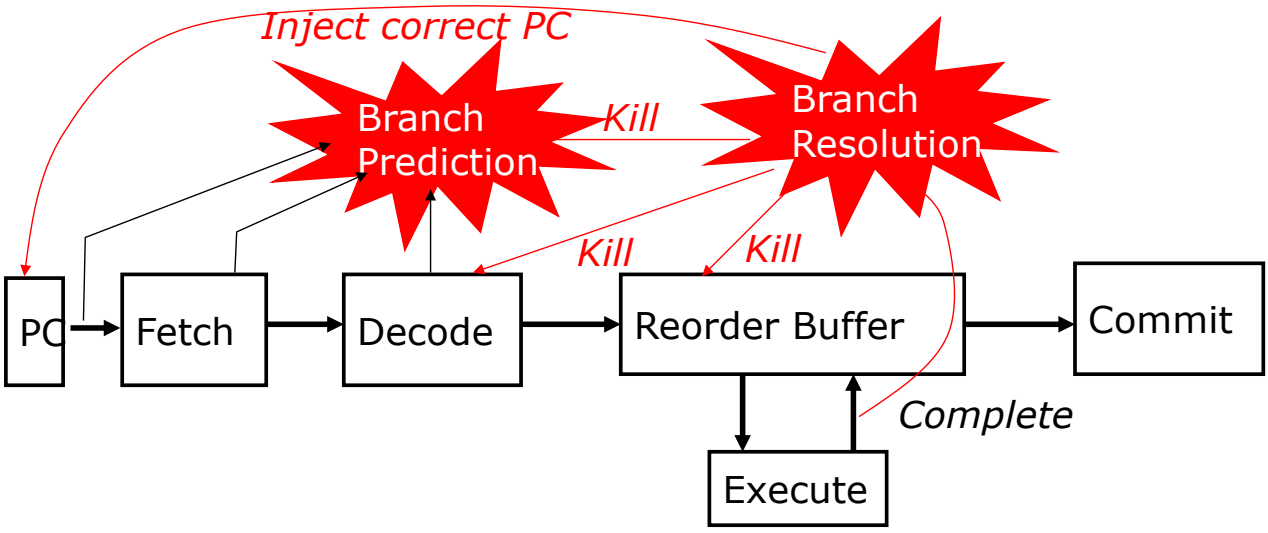
• Both styles of machine can use same branch predictors in front-end fetch pipeline, and both can execute multiple instructions per cycle

• Common to have 10-30 pipeline stages in either style of design

InO vs. OoO Mispredict Recovery

- In-order execution?
  - Design so no instruction issued after branch can write-back before branch resolves
  - Kill all instructions in pipeline behind mispredicted branch
- Out-of-order execution?
  - Multiple instructions following branch in program order can complete before branch resolves
  - A simple solution would be to handle like precise traps
    - Problem?

Branch Misprediction in Pipeline



- Can have multiple unresolved branches in ROB
- Can resolve branches out-of-order by killing all the instructions in ROB that follow a mispredicted branch
- MIPS R10K uses four mask bits to tag instructions that are dependent on up to four speculative branches
- Mask bits cleared as branch resolves, and reused for next branch

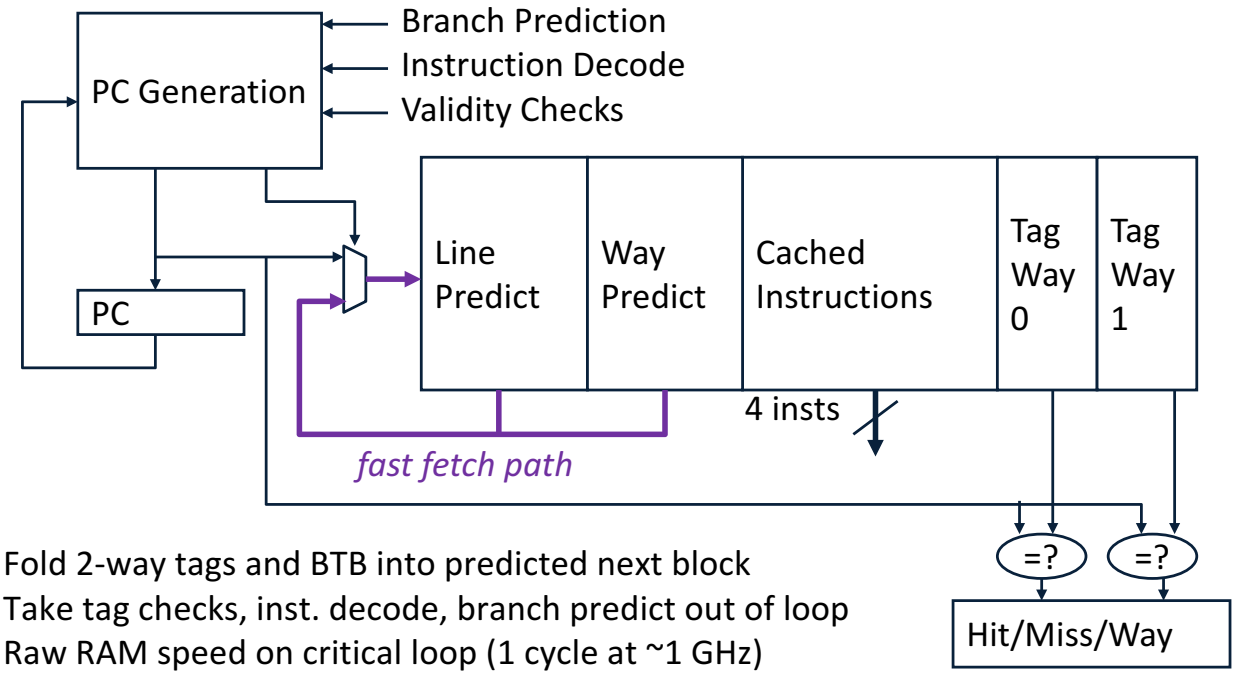
Rename Table Recovery

- Have to quickly recover rename table on branch mispredicts
- MIPS R10K only has four snapshots for each of four outstanding speculative branches
- Alpha 21264 has 80 snapshots, one per ROB instruction

Improving Instruction Fetch

- Performance of speculative out-of-order machines often limited by instruction fetch bandwidth
  - speculative execution can fetch 2-3x more instructions than are committed
  - mispredict penalties dominated by time to refill instruction window
  - taken branches are particularly troublesome

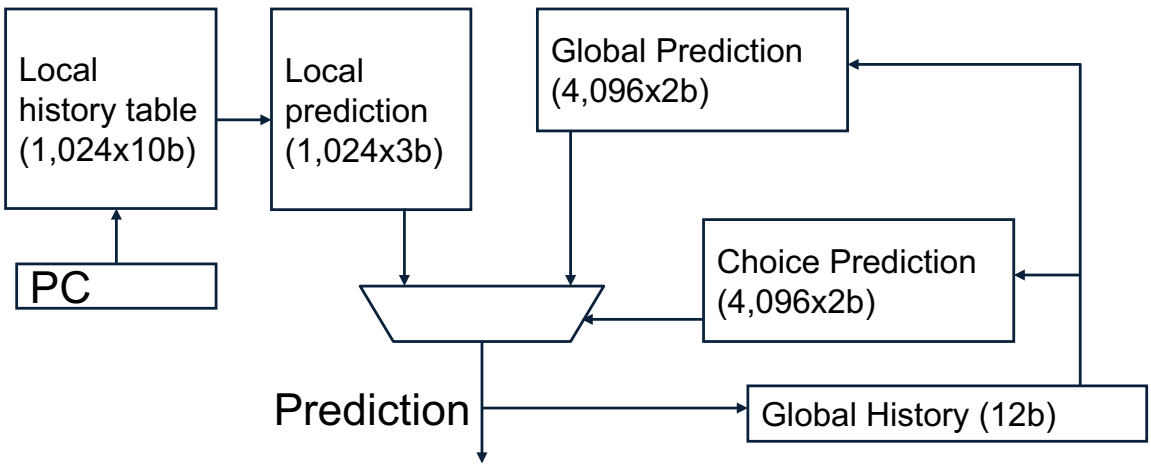
Increasing Taken Branch Bandwidth (Alpha 21264 I-Cache)



- Fold 2-way tags and BTB into predicted next block
- Take tag checks, inst. decode, branch predict out of loop
- Raw RAM speed on critical loop (1 cycle at ~1 GHz)
- 2-bit hysteresis counter per block prevents overtraining

Tournament Branch Predictor (Alpha 21264)

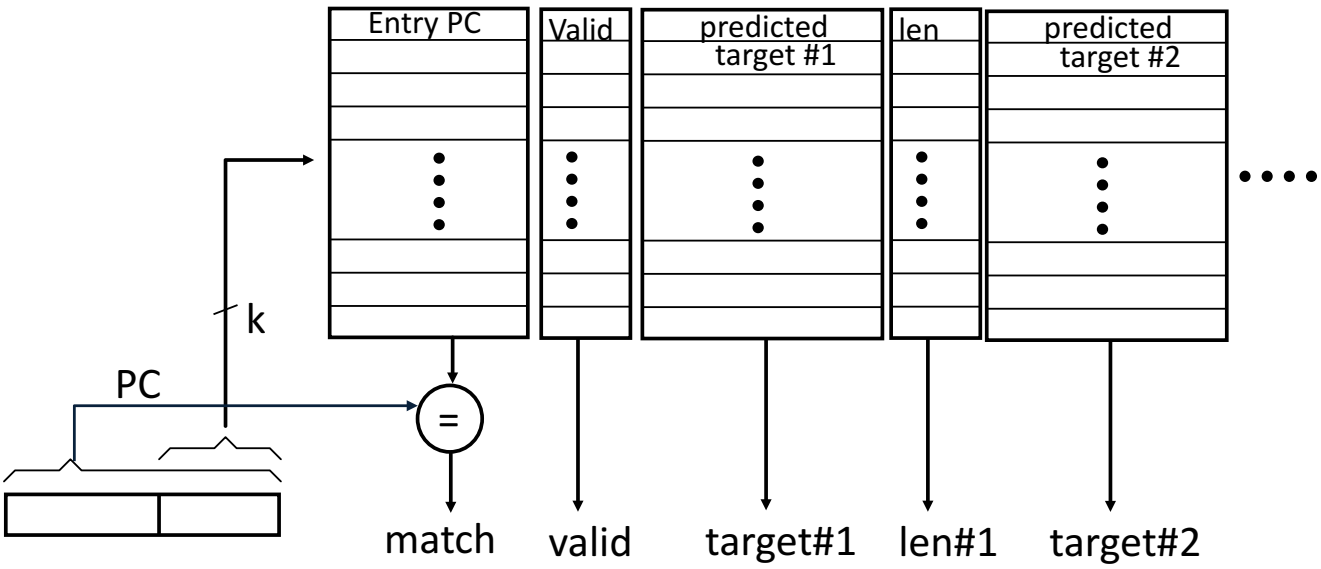
- Choice predictor learns whether best to use local or global branch history in predicting next branch
- Global history is speculatively updated but restored on mispredict
- Claim 90-100% success on range of applications



Taken Branch Limit

- Integer codes have a taken branch every 6-9 instructions
- To avoid fetch bottleneck, must execute multiple taken branches per cycle when increasing performance
- This implies:
  - predicting multiple branches per cycle
  - fetching multiple non-contiguous blocks per cycle

Branch Address Cache (Yeh, Marr, Patt)



Extend BTB to return multiple branch predictions per cycle

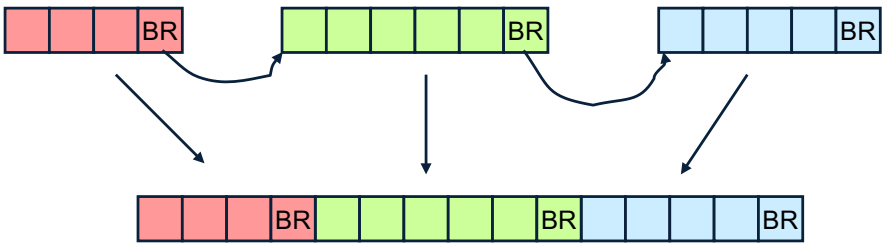
Fetching Multiple Basic Blocks

- Requires either
  - multiported cache: expensive
  - interleaving: bank conflicts will occur

- Merging multiple blocks to feed to decoders adds latency increasing mispredict penalty and reducing branch throughput

Trace Cache

- Key Idea: Pack multiple non-contiguous basic blocks into one contiguous trace cache line

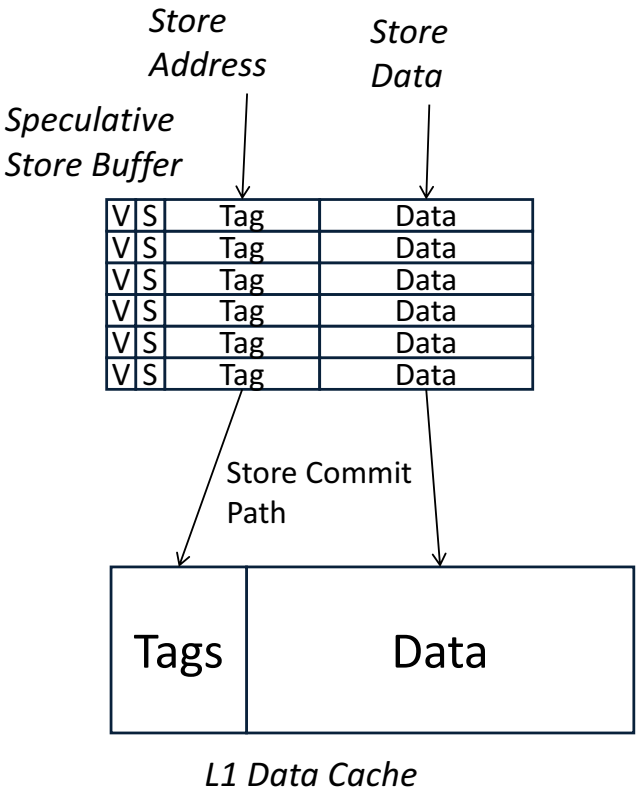


- Single fetch brings in multiple basic blocks
- Trace cache indexed by start address *and* next *n* branch predictions
- Used in Intel Pentium-4 processor to hold decoded uops

Load-Store Queue Design

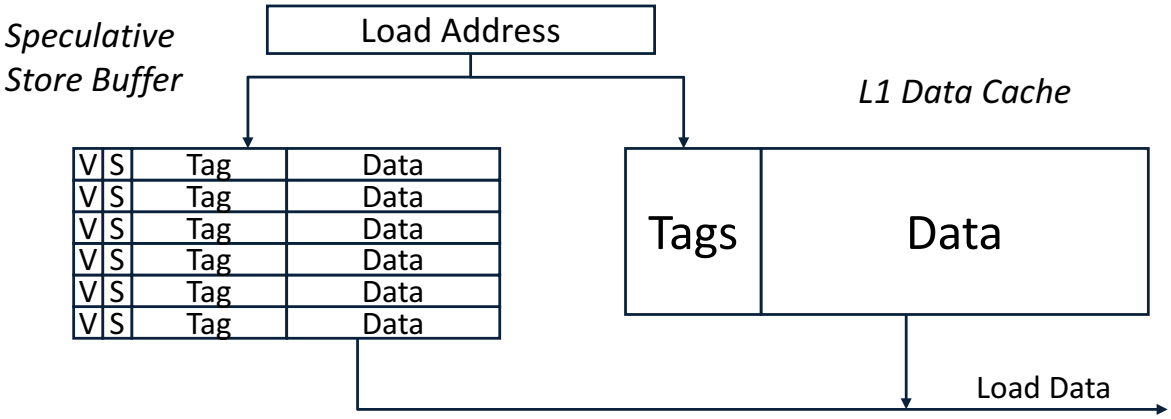
- After control hazards, data hazards through memory are probably next most important bottleneck to superscalar performance
- Modern superscalars use very sophisticated load-store reordering techniques to reduce effective memory latency by allowing loads to be speculatively issued

Speculative Store Buffer



- Just like register updates, stores should not modify the memory until after the instruction is committed. A speculative store buffer is a structure introduced to hold speculative store data.
- During decode, store buffer slot allocated in program order
- Stores split into "store address" and "store data" micro-operations
- "Store address" execution writes tag
- "Store data" execution writes data
- Store commits when oldest instruction and both address and data available:
  - clear speculative bit and eventually move data to cache
- On store abort:
  - clear valid bit

Load bypass from speculative store buffer



- If data in both store buffer and cache, which should we use?
  Speculative store buffer
- If same address in store buffer twice, which should we use?
  Youngest store older than load

Memory Dependencies

```
sd x1, (x2)
ld x3, (x4)
```

- When can we execute the load?

In-Order Memory Queue

- Execute all loads and stores in program order
- => Load and store cannot leave ROB for execution until all previous loads and stores have completed execution
- Can still execute loads and stores speculatively, and out-of-order with respect to other instructions
- Need a structure to handle memory ordering…

Conservative O-o-O Load Execution

```
sd x1, (x2)
ld x3, (x4)
```

- Can execute load before store, if addresses known and **x4** != **x2**
- Each load address compared with addresses of all previous uncommitted stores
  - can use partial conservative check i.e., bottom 12 bits of address, to save hardware
- Don't execute load if any previous store address not known
- (MIPS R10K, 16-entry address queue)

Address Speculation

```
sd x1, (x2)
ld x3, (x4)
```

- Guess that **x4** != **x2**
- Execute load before store address known
- Need to hold all completed but uncommitted load/store addresses in program order
- If subsequently find **x4==x2**, squash load and all following instructions

- => Large penalty for inaccurate address speculation

Memory Dependence Prediction (Alpha 21264)

```
sd x1, (x2)
ld x3, (x4)
```

- Guess that **x4** != **x2** and execute load before store

- If later find **x4**==**x2**, squash load and all following instructions, but mark load instruction as store-wait

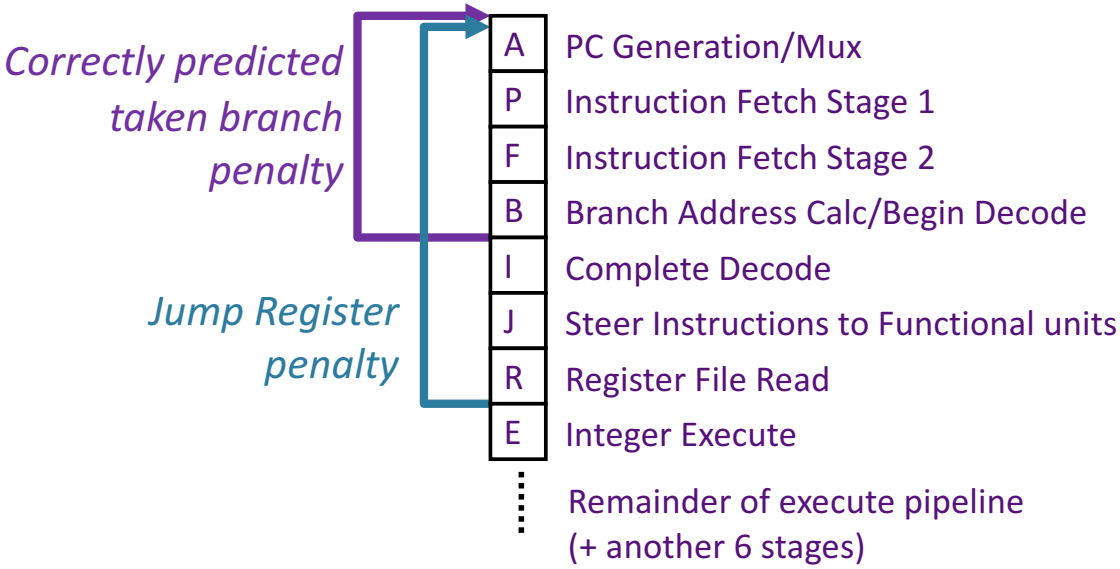
- Subsequent executions of the same load instruction will wait for all previous stores to complete

- Periodically clear store-wait bits

Part III: More Advanced Out-of-Order Superscalar Designs

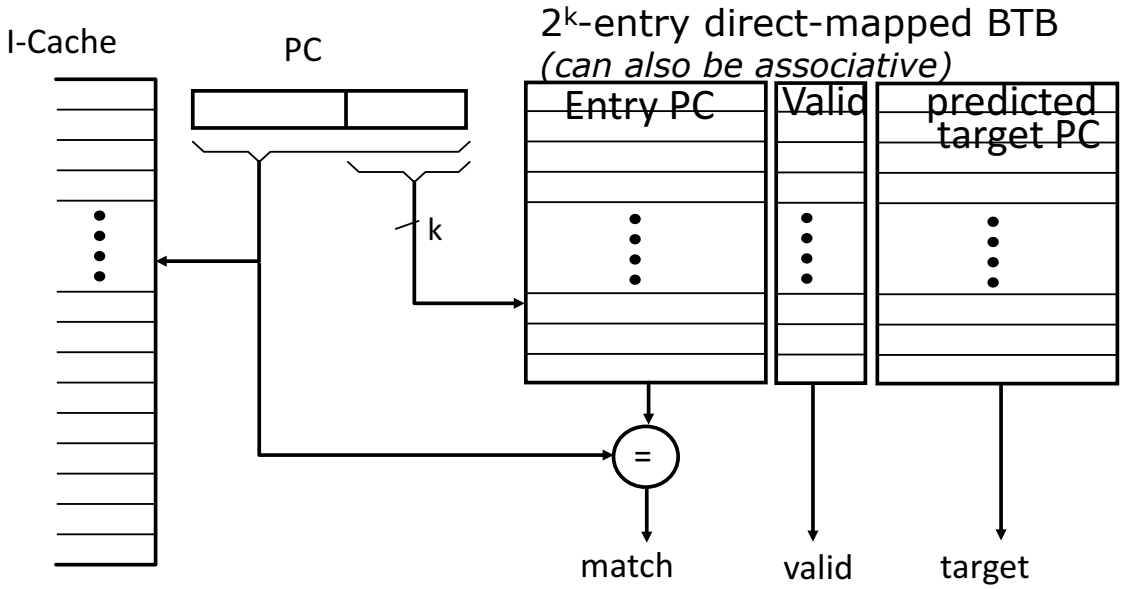
Limitations of BHTs

Only predicts branch direction. Therefore, cannot redirect fetch stream until after branch target is determined.



UltraSPARC-III fetch pipeline

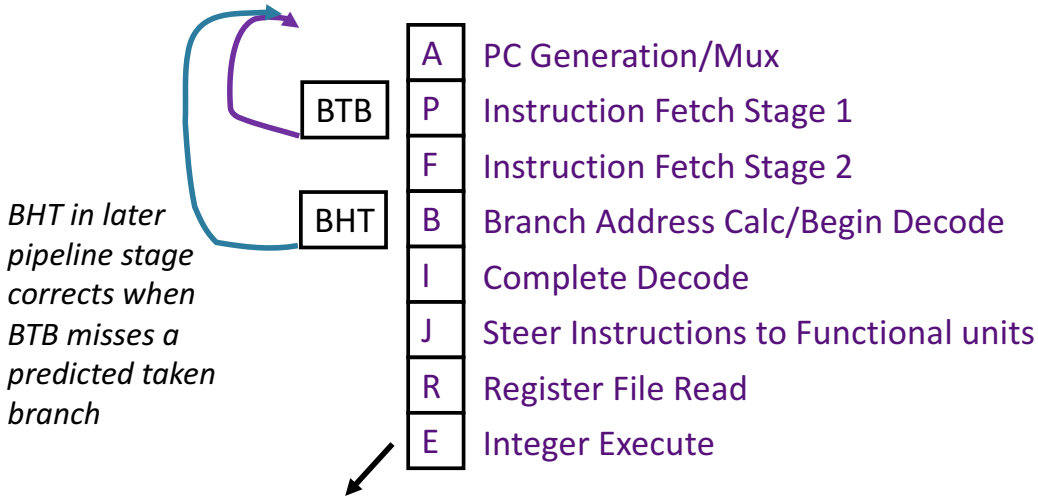
Branch Target Buffer (BTB)



- Keep both the branch PC and target PC in the BTB
- PC+4 is fetched if match fails
- Only *taken* branches and jumps held in BTB
- Next PC determined *before* branch fetched and decoded

Combining BTB and BHT

- BTB entries are considerably more expensive than BHT, but can redirect fetches at earlier stage in pipeline and can accelerate indirect branches (JR)
- BHT can hold many more entries and is more accurate



BTB/BHT only updated after branch resolves in E stage

Uses of Jump Register (JR)

- Switch statements (jump to address of matching case)

  BTB works well if same case used repeatedly

- Dynamic function call (jump to run-time function address)

  BTB works well if same function usually called, (e.g., in C++ programming, when objects have same type in virtual function call)

- Subroutine returns (jump to return address)

  BTB works well if usually return to the same place

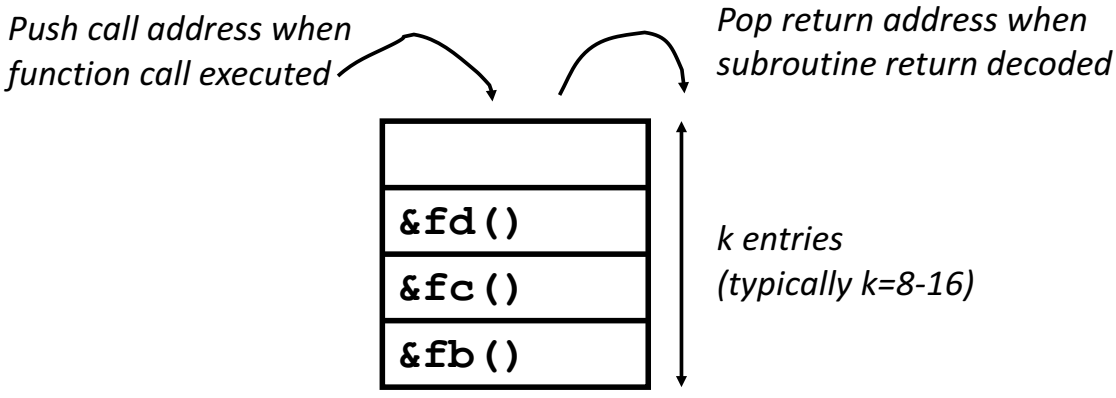
  ⇒ *Often one function called from many distinct call sites!*

How well does BTB work for each of these cases?

Subroutine Return Stack

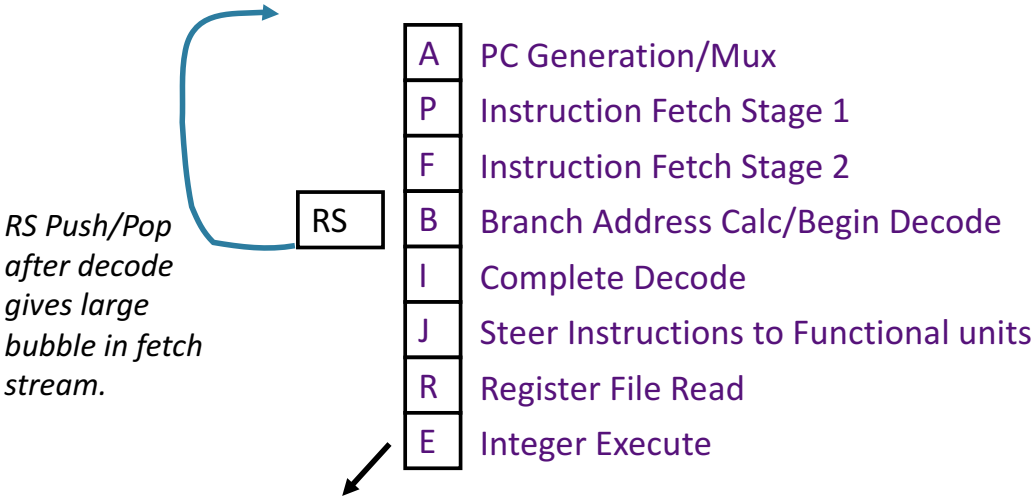
Small structure to accelerate JR for subroutine returns, typically much more accurate than BTBs.

```
fa() { fb(); }
fb() { fc(); }
fc() { fd(); }
```



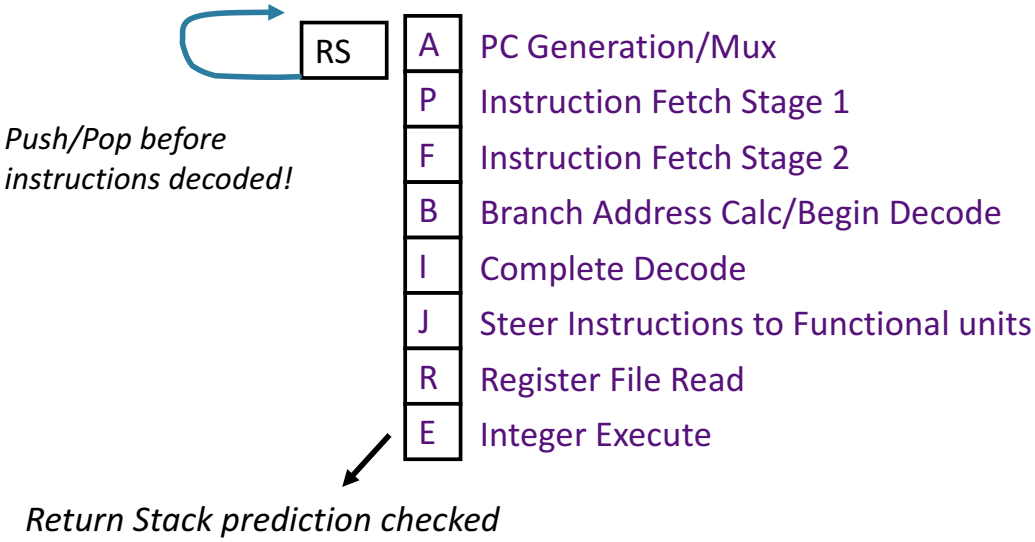
Return Stack in Pipeline

- How to use return stack (RS) in deep fetch pipeline?
- Only know if subroutine call/return at decode

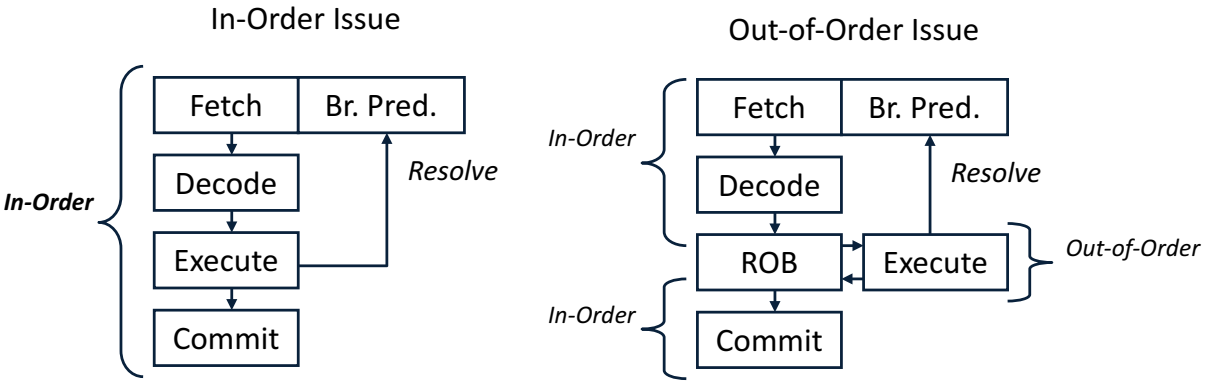


Return Stack in Pipeline

- Can remember whether PC is subroutine call/return using BTB-like structure
- Instead of target-PC, just store push/pop bit



In-Order vs. Out-of-Order Branch Prediction



- Speculative fetch but not speculative execution - branch resolves before later instructions complete
- Completed values held in bypass network until commit

- Speculative execution, with branches resolved after later instructions complete
- Completed values held in rename registers in ROB or unified physical register file until commit

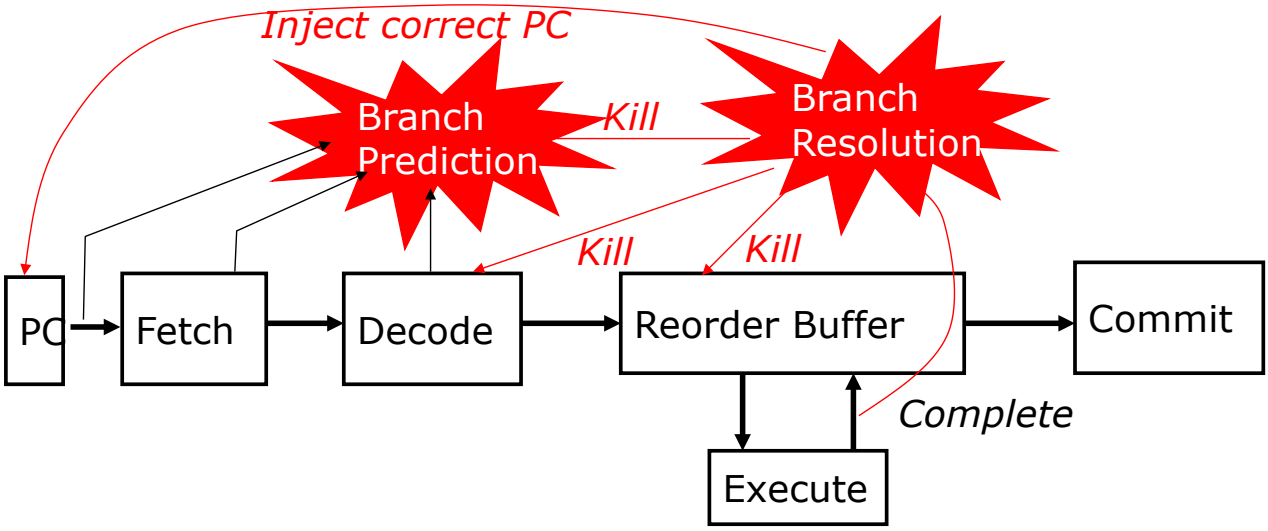
• Both styles of machine can use same branch predictors in front-end fetch pipeline, and both can execute multiple instructions per cycle

• Common to have 10-30 pipeline stages in either style of design

InO vs. OoO Mispredict Recovery

- In-order execution?
  - Design so no instruction issued after branch can write-back before branch resolves
  - Kill all instructions in pipeline behind mispredicted branch
- Out-of-order execution?
  - Multiple instructions following branch in program order can complete before branch resolves
  - A simple solution would be to handle like precise traps
    - Problem?

Branch Misprediction in Pipeline



- Can have multiple unresolved branches in ROB
- Can resolve branches out-of-order by killing all the instructions in ROB that follow a mispredicted branch
- MIPS R10K uses four mask bits to tag instructions that are dependent on up to four speculative branches
- Mask bits cleared as branch resolves, and reused for next branch

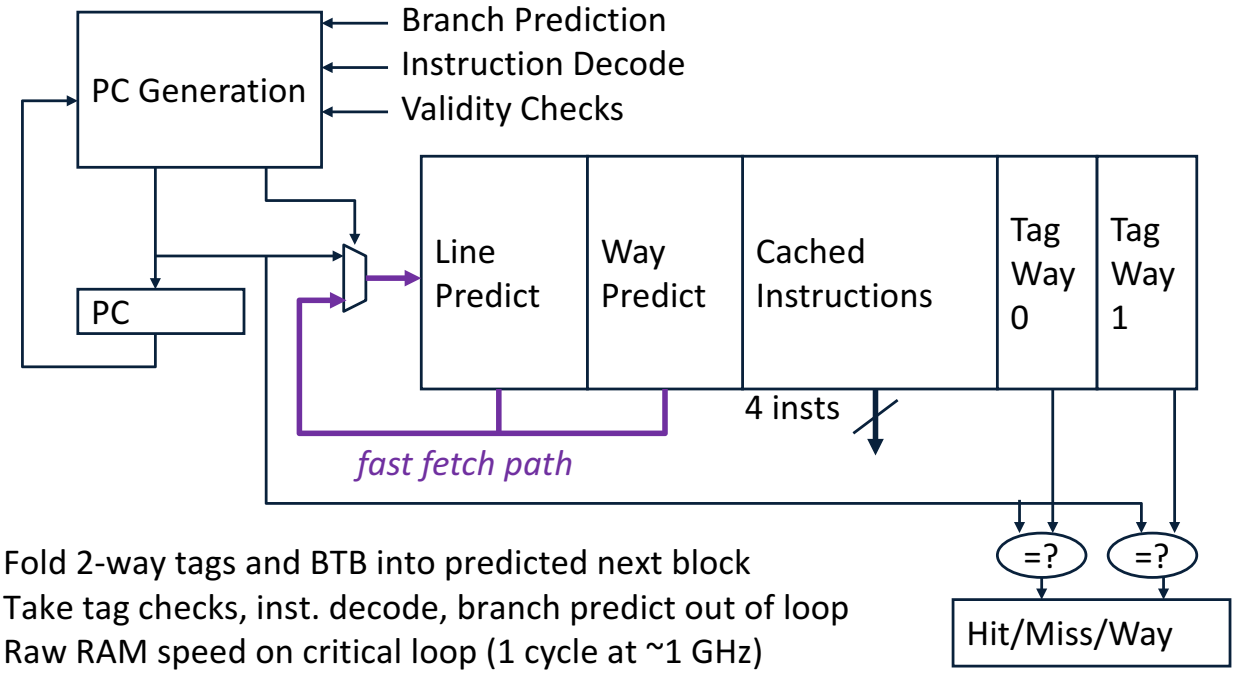
Rename Table Recovery

- Have to quickly recover rename table on branch mispredicts
- MIPS R10K only has four snapshots for each of four outstanding speculative branches
- Alpha 21264 has 80 snapshots, one per ROB instruction

Improving Instruction Fetch

- Performance of speculative out-of-order machines often limited by instruction fetch bandwidth
  - speculative execution can fetch 2-3x more instructions than are committed
  - mispredict penalties dominated by time to refill instruction window
  - taken branches are particularly troublesome

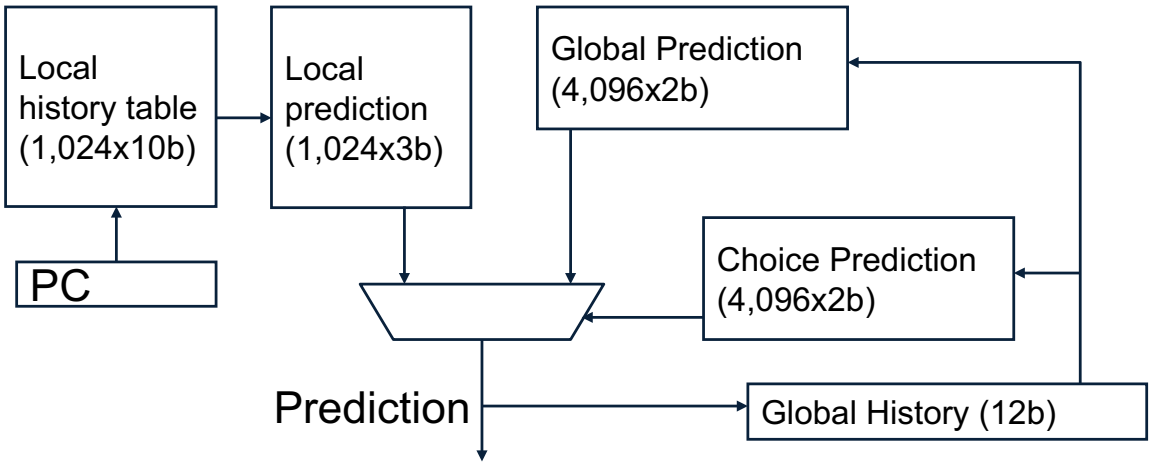
Increasing Taken Branch Bandwidth (Alpha 21264 I-Cache)



- Fold 2-way tags and BTB into predicted next block
- Take tag checks, inst. decode, branch predict out of loop
- Raw RAM speed on critical loop (1 cycle at ~1 GHz)
- 2-bit hysteresis counter per block prevents overtraining

Tournament Branch Predictor (Alpha 21264)

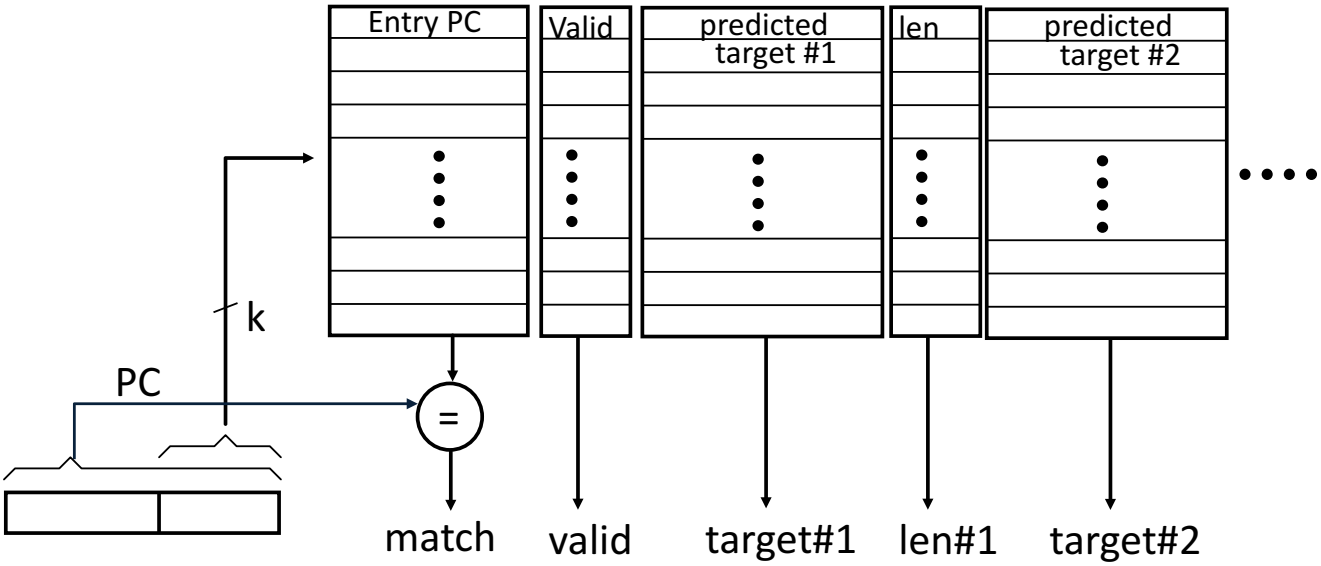
- Choice predictor learns whether best to use local or global branch history in predicting next branch
- Global history is speculatively updated but restored on mispredict
- Claim 90-100% success on range of applications



Taken Branch Limit

- Integer codes have a taken branch every 6-9 instructions
- To avoid fetch bottleneck, must execute multiple taken branches per cycle when increasing performance
- This implies:
  - predicting multiple branches per cycle
  - fetching multiple non-contiguous blocks per cycle

Branch Address Cache
(Yeh, Marr, Patt)



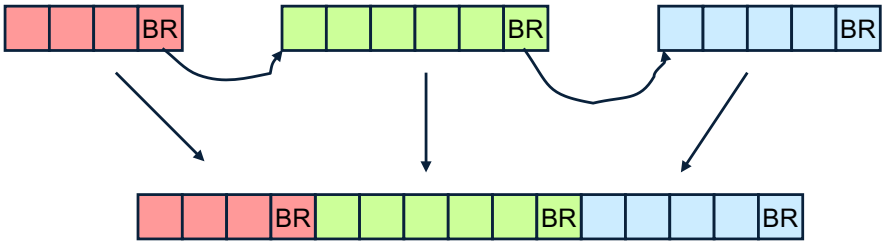
Extend BTB to return multiple branch predictions per cycle

Fetching Multiple Basic Blocks

- Requires either
  - multiported cache: expensive
  - interleaving: bank conflicts will occur
- Merging multiple blocks to feed to decoders adds latency increasing mispredict penalty and reducing branch throughput

Trace Cache

- Key Idea: Pack multiple non-contiguous basic blocks into one contiguous trace cache line

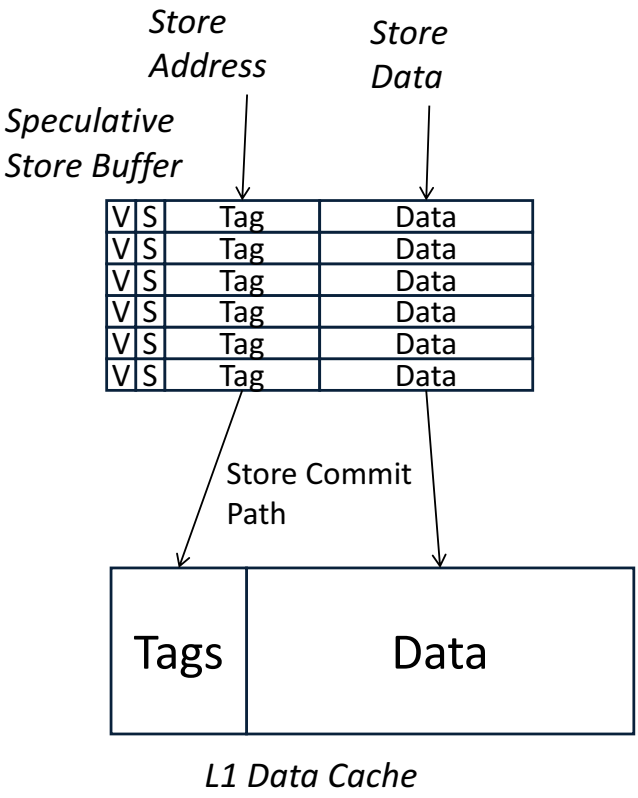


- Single fetch brings in multiple basic blocks
- Trace cache indexed by start address *and* next *n* branch predictions
- Used in Intel Pentium-4 processor to hold decoded uops

Load-Store Queue Design

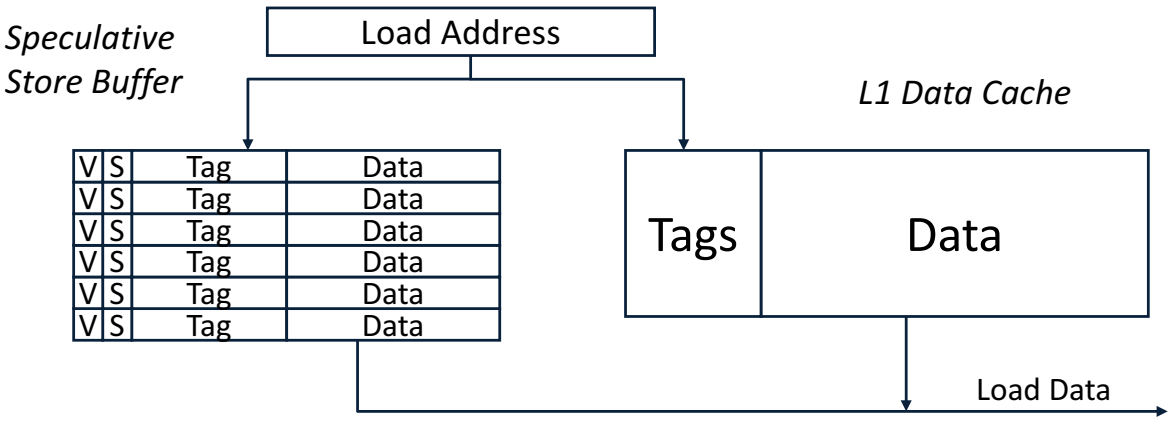
- After control hazards, data hazards through memory are probably next most important bottleneck to superscalar performance
- Modern superscalars use very sophisticated load-store reordering techniques to reduce effective memory latency by allowing loads to be speculatively issued

Speculative Store Buffer



- Just like register updates, stores should not modify the memory until after the instruction is committed. A speculative store buffer is a structure introduced to hold speculative store data.
- During decode, store buffer slot allocated in program order
- Stores split into "store address" and "store data" micro-operations
- "Store address" execution writes tag
- "Store data" execution writes data
- Store commits when oldest instruction and both address and data available:
  - clear speculative bit and eventually move data to cache
- On store abort:
  - clear valid bit

Load bypass from speculative store buffer



- If data in both store buffer and cache, which should we use?
  Speculative store buffer
- If same address in store buffer twice, which should we use?
  Youngest store older than load

Memory Dependencies

```
sd x1, (x2)
ld x3, (x4)
```

- When can we execute the load?

In-Order Memory Queue

- Execute all loads and stores in program order
- => Load and store cannot leave ROB for execution until all previous loads and stores have completed execution
- Can still execute loads and stores speculatively, and out-of-order with respect to other instructions
- Need a structure to handle memory ordering…

Conservative O-o-O Load Execution

```
sd x1, (x2)
ld x3, (x4)
```

- Can execute load before store, if addresses known and **x4** != **x2**
- Each load address compared with addresses of all previous uncommitted stores
  - can use partial conservative check i.e., bottom 12 bits of address, to save hardware
- Don't execute load if any previous store address not known
- (MIPS R10K, 16-entry address queue)

Address Speculation

```
sd x1, (x2)
ld x3, (x4)
```

- Guess that **x4** != **x2**
- Execute load before store address known
- Need to hold all completed but uncommitted load/store addresses in program order
- If subsequently find **x4==x2**, squash load and all following instructions

- => Large penalty for inaccurate address speculation

Memory Dependence Prediction (Alpha 21264)

```
sd x1, (x2)
ld x3, (x4)
```

- Guess that **x4** != **x2** and execute load before store
- If later find **x4**==**x2**, squash load and all following instructions, but mark load instruction as store-wait
- Subsequent executions of the same load instruction will wait for all previous stores to complete
- Periodically clear store-wait bits

Acknowledgements

- These course notes were developed by:
  - Krste Asanovic (UCB)
  - Arvind (MIT)
  - Joel Emer (Intel/MIT)
  - James Hoe (CMU)
  - John Kubiatowicz (UCB)
  - David Patterson (UCB)