ARCHITECTURE OF COMPUTER SYSTEMS LECTURE 12 - ADVANCED OUT-OF-ORDER SUPERSCALARS

LAST TIME IN LECTURE 11

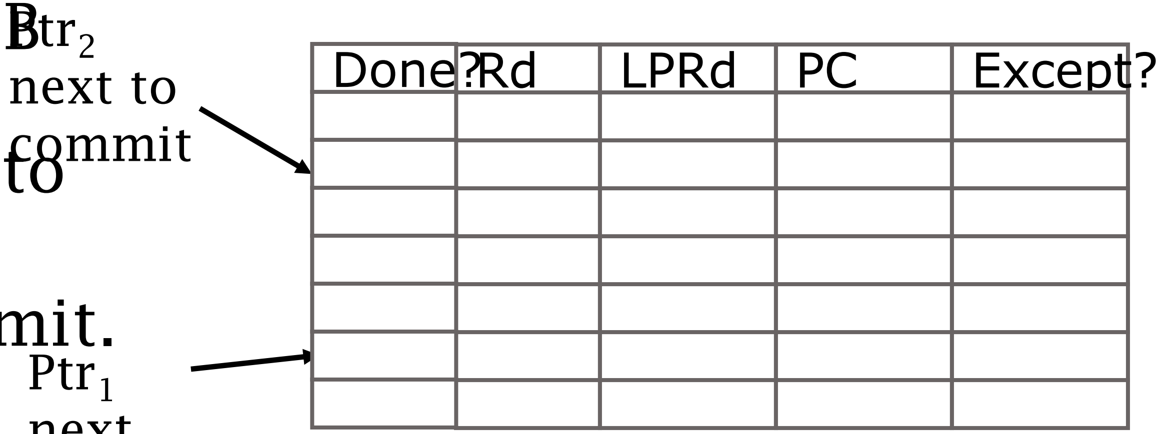
- Register renaming removes WAR, WAW hazards
- In-order fetch/decode, out-of-order execute, in-order commit gives high performance and precise exceptions
- Need to rapidly recover on branch mispredictions
- Unified physical register file machines remove data values from ROB
  - All values only read and written during execution
  - Only register tags held in ROB

SEPARATE PENDING INSTRUCTION WINDOW FROM ROB

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

us	ex	op	p1	PR1	p2	PR2	PRd	ROB#

Reorder buffer used to hold exception information for commit.



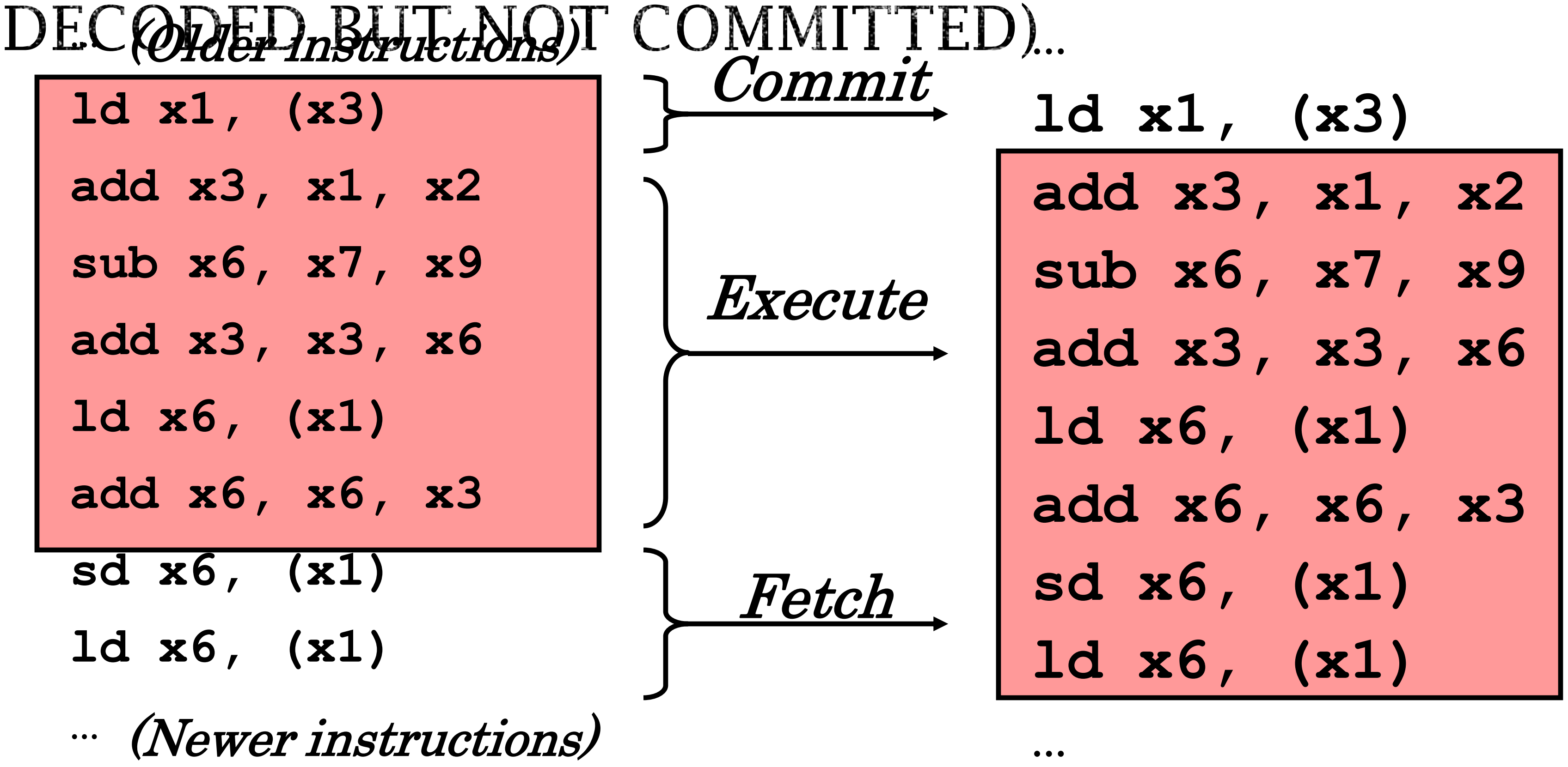
Ptr_2 next to commit

Done?	Rd	LPRd	PC	Except?

Ptr_1 next available

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

REORDER BUFFER HOLDS ACTIVE INSTRUCTIONS (DECODED BUT NOT COMMITTED)



… *(Older instructions)*

```
ld x1, (x3)
add x3, x1, x2
sub x6, x7, x9
add x3, x3, x6
ld x6, (x1)
add x6, x6, x3
sd x6, (x1)
ld x6, (x1)
```

… *(Newer instructions)*

Commit

Execute

Fetch

…

```
ld x1, (x3)
add x3, x1, x2
sub x6, x7, x9
add x3, x3, x6
ld x6, (x1)
add x6, x6, x3
sd x6, (x1)
ld x6, (x1)
```

…

Cycle *t*

Cycle *t + 1*

ISSUE TIMING

i1	Add R1,R1,#1	Issue$_1$	Execute$_1$		
i2	Sub R1,R1,#1			Issue$_2$	Execute$_2$

How can we issue earlier?

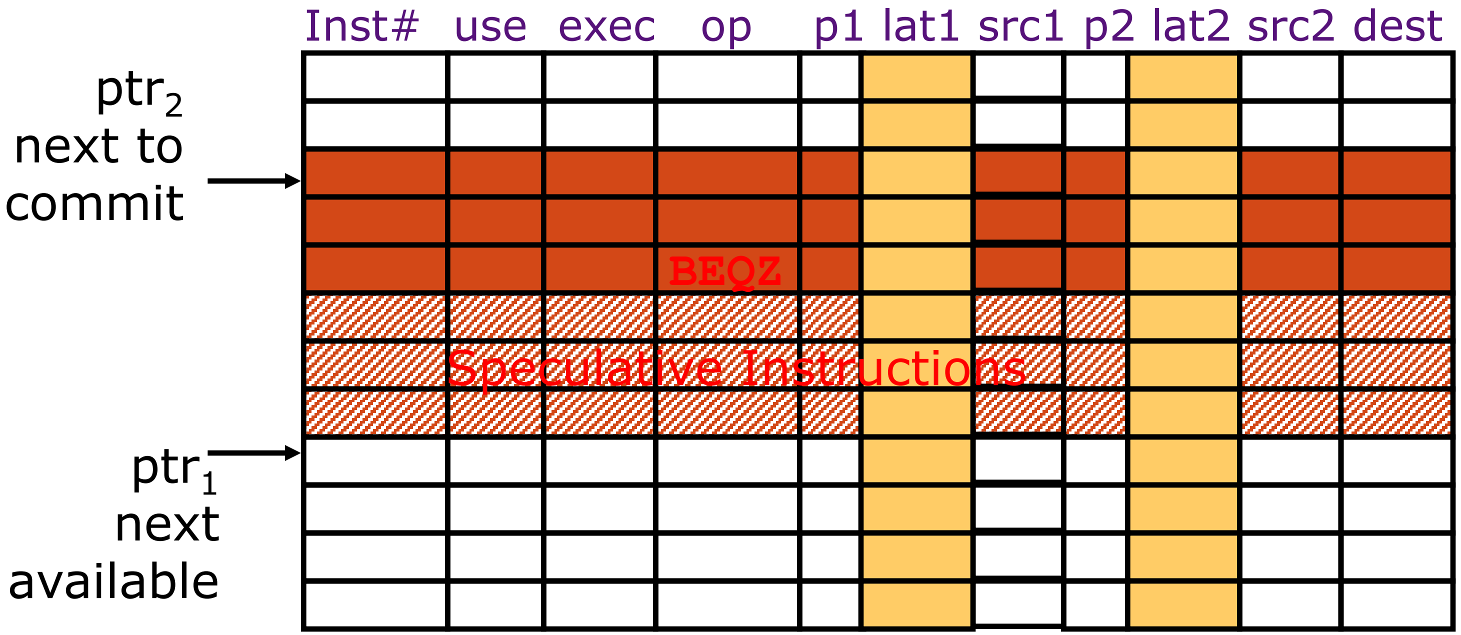
Using knowledge of execution latency (bypass)

i1	Add R1,R1,#1	Issue$_1$	Execute$_1$		
i2	Sub R1,R1,#1		Issue$_2$	Execute$_2$	

What makes this schedule fail?

If execution latency wasn't as expected

ISSUE QUEUE WITH LATENCY PREDICTION



Issue Queue (Reorder buffer)

- Fixed latency: latency included in queue entry (‘bypassed’)
- Predicted latency: latency included in queue entry (speculated)
- Variable latency: wait for completion signal (stall)

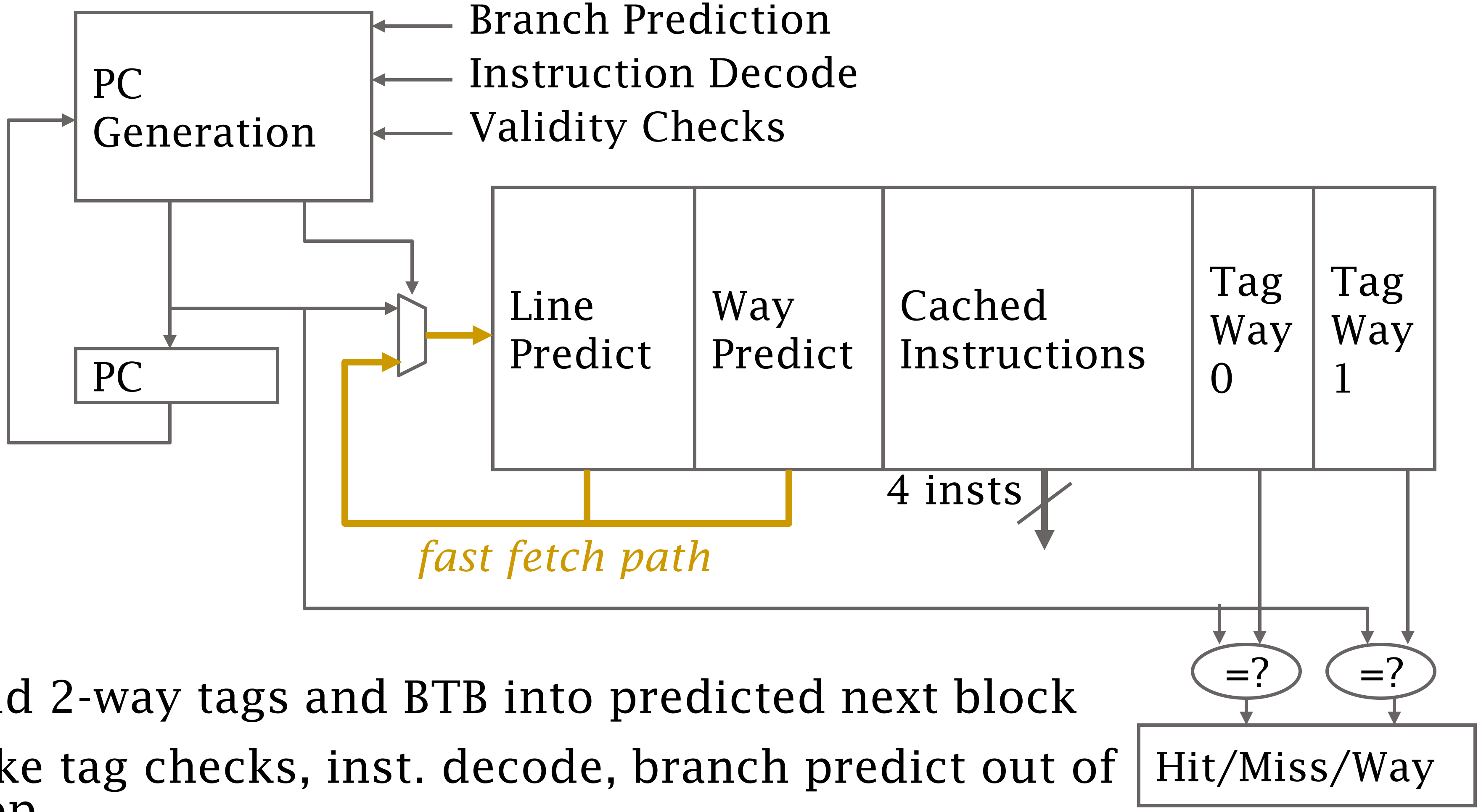
IMPROVING INSTRUCTION FETCH

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)



Branch Prediction

Instruction Decode

Validity Checks

PC Generation

PC

Line Predict

Way Predict

Cached Instructions

Tag Way 0

Tag Way 1

4 insts

fast fetch path

=?

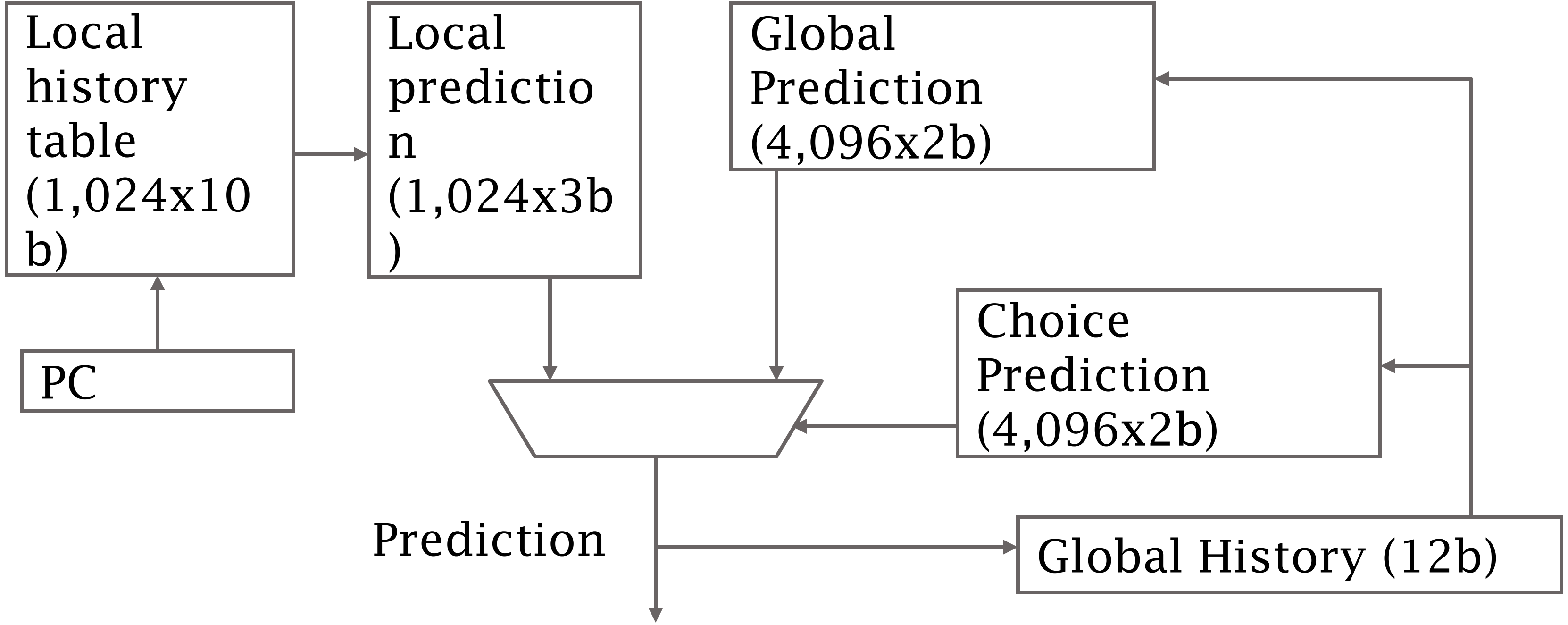
=?

Hit/Miss/Way

- 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)



Local history table (1,024x10b)

Local prediction (1,024x3b)

Global Prediction (4,096x2b)

PC

Choice Prediction (4,096x2b)

Prediction

Global History (12b)

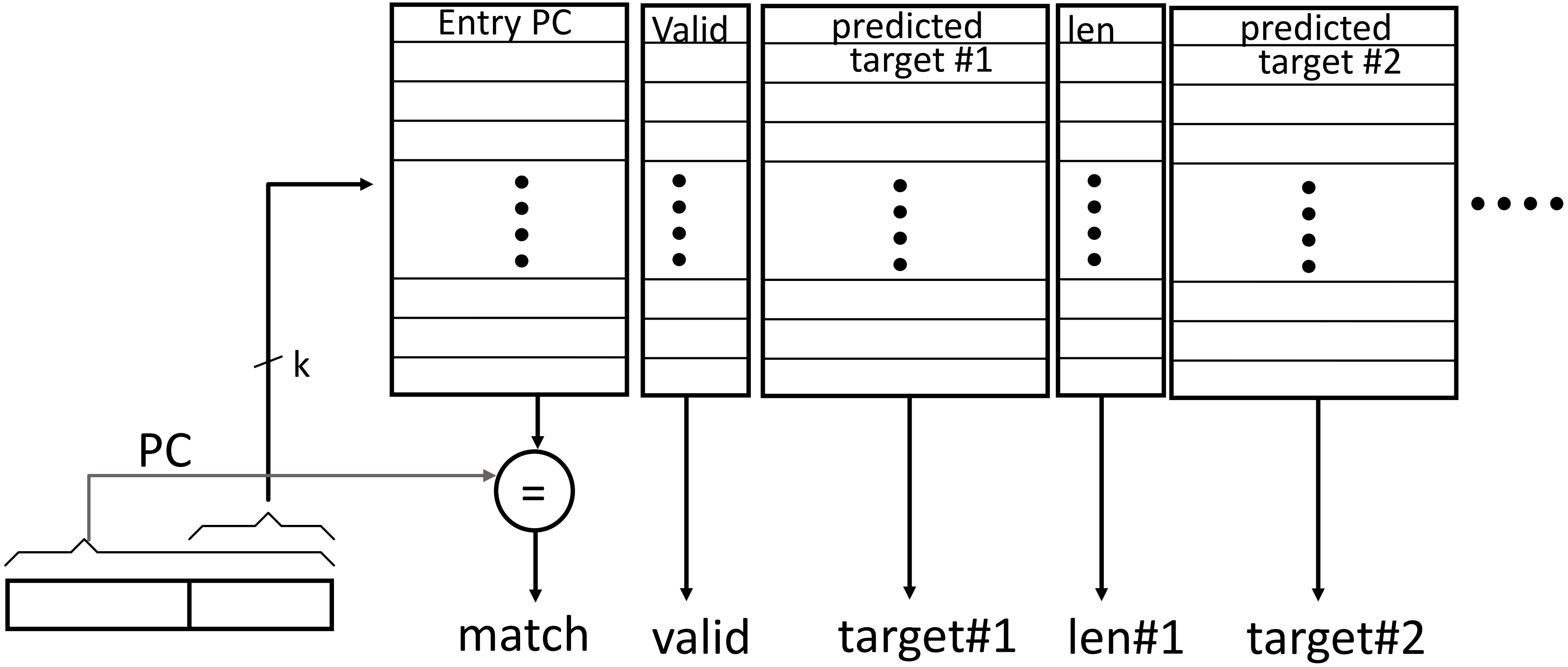
- Choice predictor learns whether best to use local or global branch history in predicting next branch
- Global history is speculatively updated but restored on mispredict
- Claim 90-100% success on range of applications

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

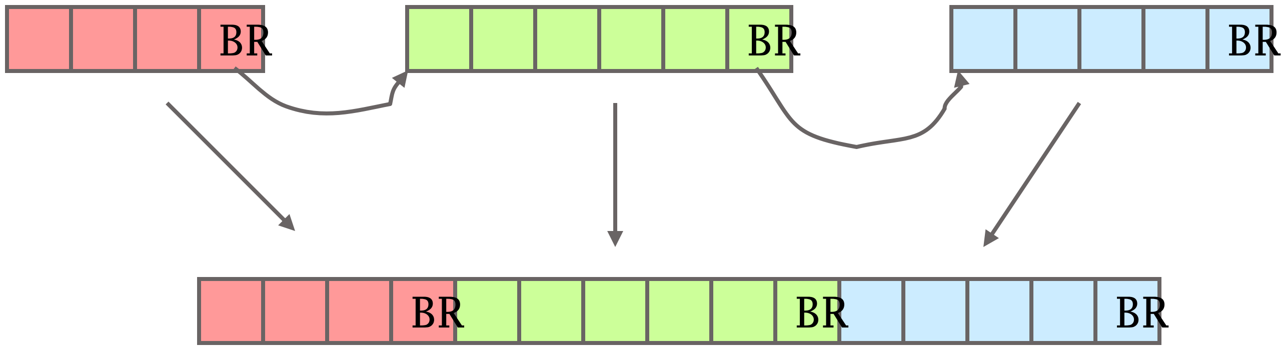
Requires either

- 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

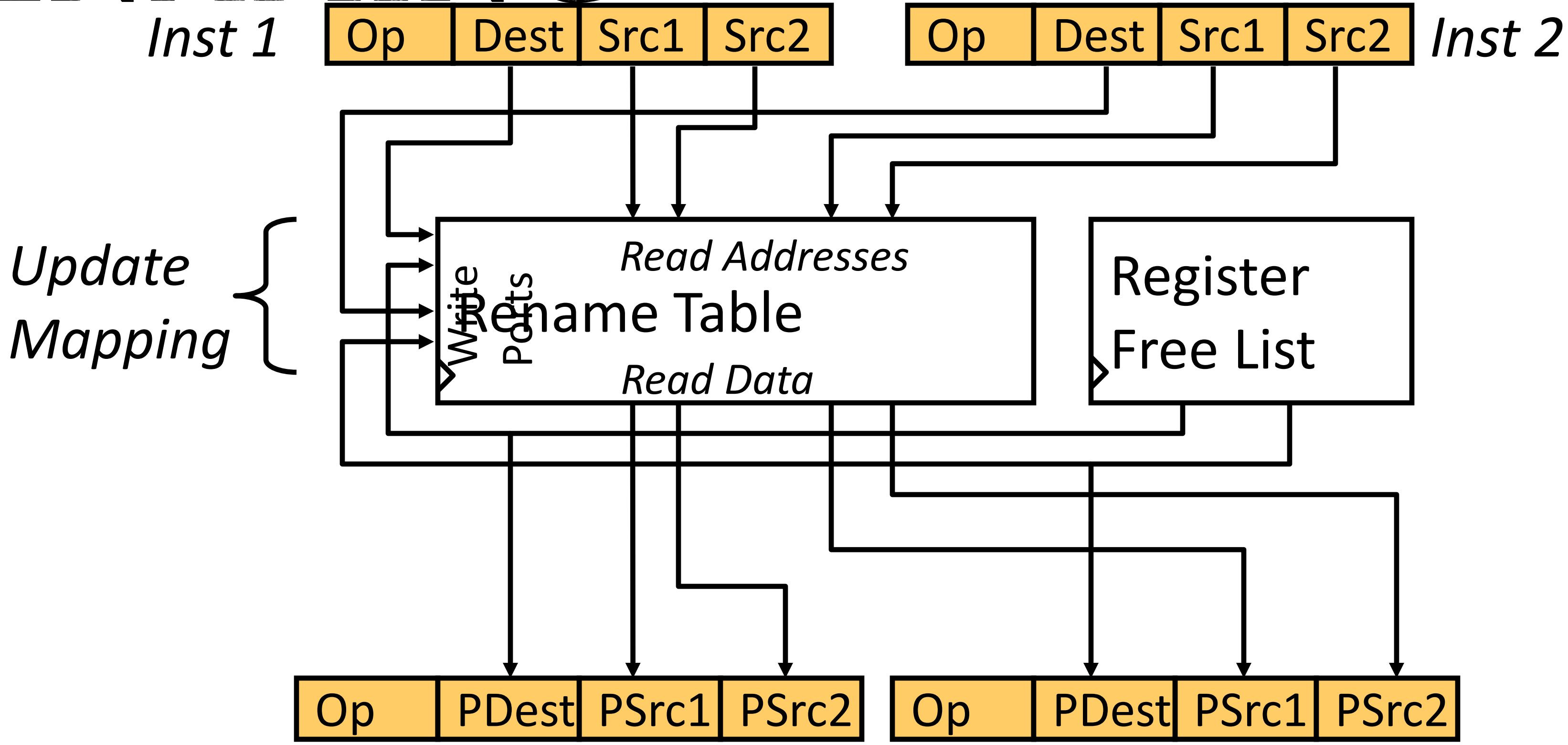
Key Idea: Pack multiple non-contiguous basic blocks into one contiguous trace cache line



- 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

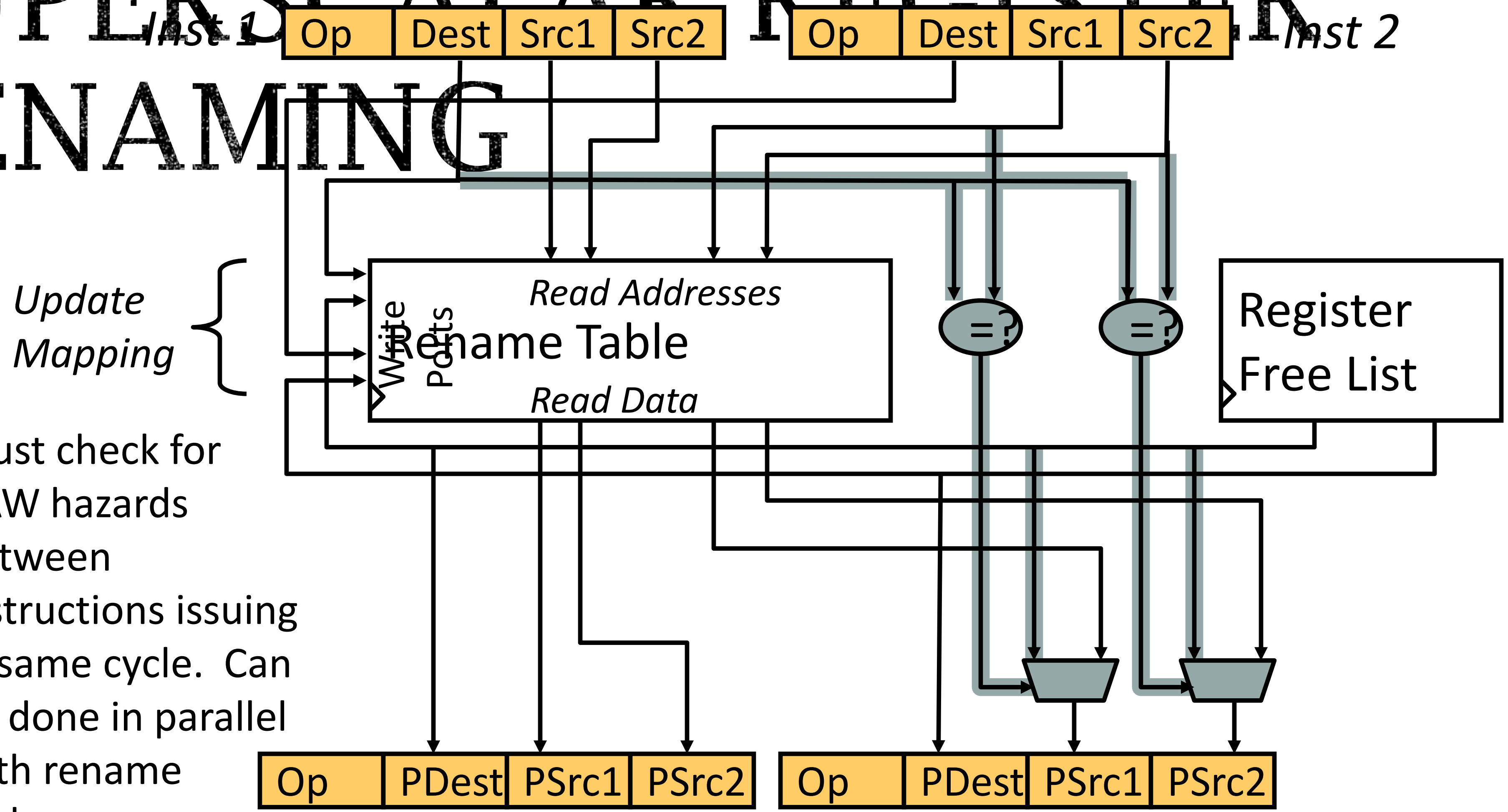
SUPERSCALAR REGISTER RENAMING

- 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.

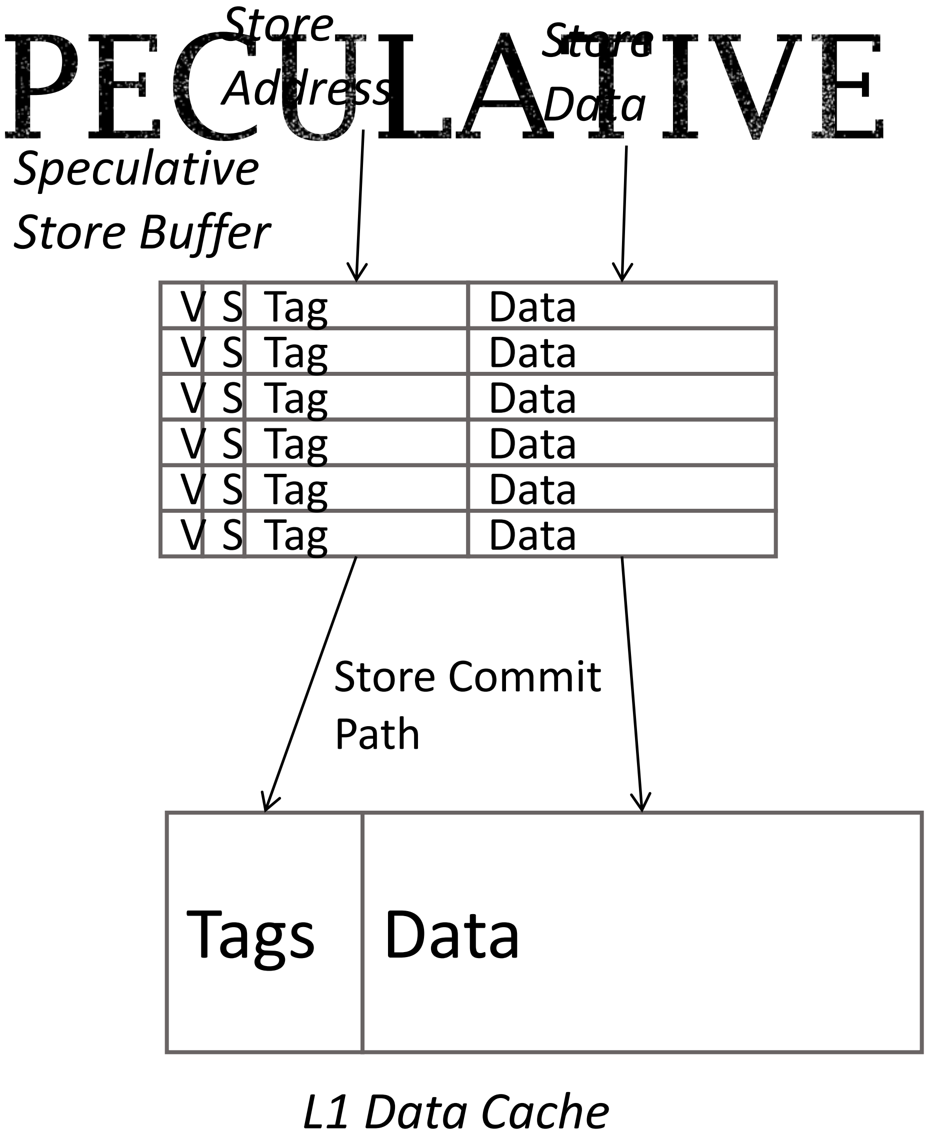
MIPS R10K renames 4 serially-RAW-dependent insts/cycle

SPECULATIVE LOADS / STORES

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.

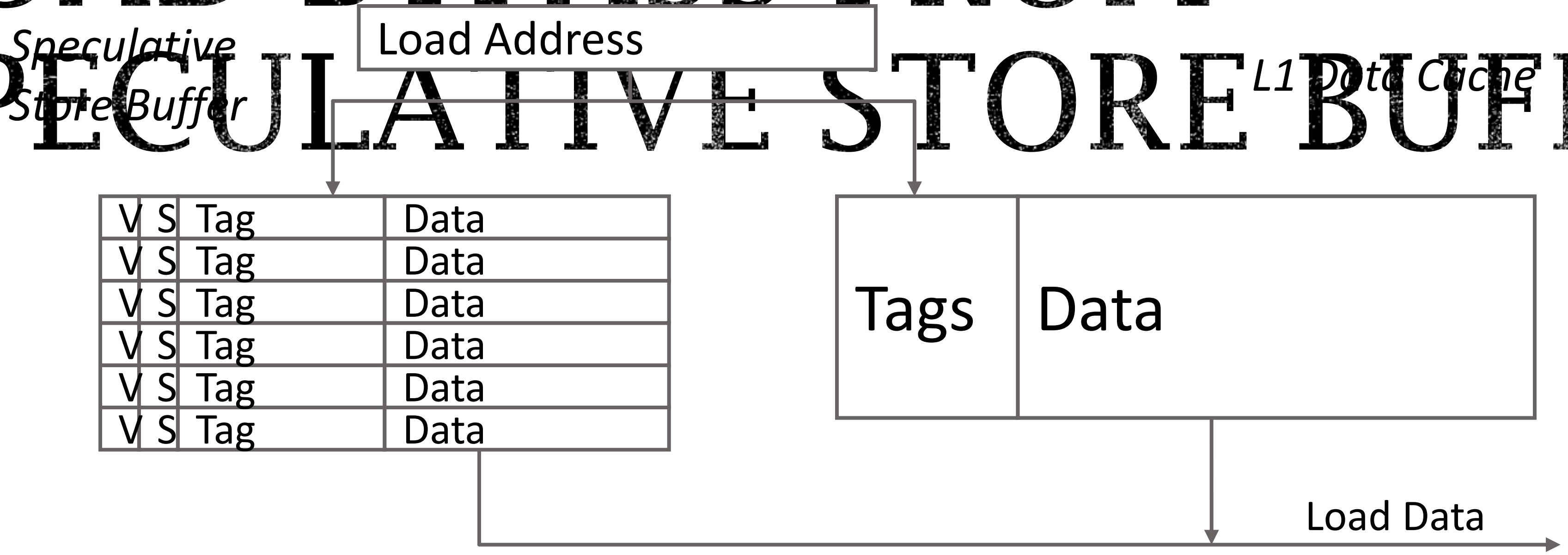
SPECULATIVE STORE BUFFER



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" execute writes tag
- "Store data" execute 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



Speculative Store Buffer

Load Address

L1 Data Cache

V	S	Tag	Data
V	S	Tag	Data
V	S	Tag	Data
V	S	Tag	Data
V	S	Tag	Data
V	S	Tag	Data

Tags	Data

Load Data

- 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

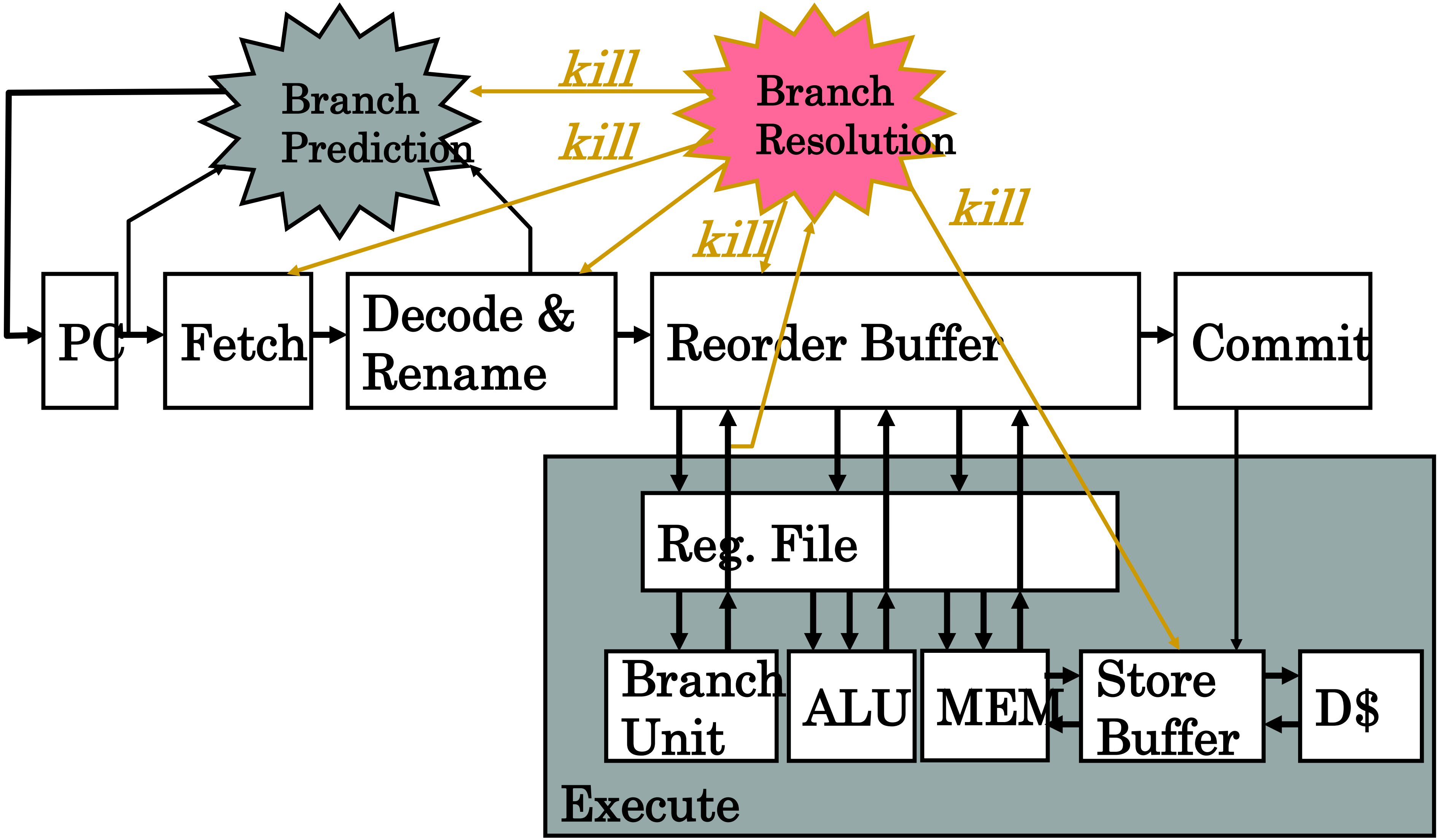
MEMORY DEPENDENCE PREDICTION

(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

Datapath: Branch Prediction and Speculative Execution



INSTRUCTION FLOW IN UNIFIED PHYSICAL REGISTER FILE PIPELINE

- Fetch
  - Get instruction bits from current guess at PC, place in fetch buffer
  - Update PC using sequential address or branch predictor (BTB)
- Decode/Rename
  - Take instruction from fetch buffer
  - Allocate resources to execute instruction:
    - Destination physical register, if instruction writes a register
    - Entry in reorder buffer to provide in-order commit
    - Entry in issue window to wait for execution
    - Entry in memory buffer, if load or store
  - Decode will stall if resources not available
  - Rename source and destination registers
  - Check source registers for readiness
  - Insert instruction into issue window+reorder buffer+memory buffer

MEMORY INSTRUCTIONS

- Split store instruction into two pieces during decode:
    - Address calculation, store-address
    - Data movement, store-data
- Allocate space in program order in memory buffers during decode
- Store instructions:
    - Store-address calculates address and places in store buffer
    - Store-data copies store value into store buffer
    - Store-address and store-data execute independently out of issue window
    - Stores only commit to data cache at commit point
- Load instructions:
    - Load address calculation executes from window
    - Load with completed effective address searches memory buffer
    - Load instruction may have to wait in memory buffer for earlier store ops to resolve

ISSUE STAGE

- Writebacks from completion phase “wakeup” some instructions by causing their source operands to become ready in issue window
  - In more speculative machines, might wake up waiting loads in memory buffer
- Need to “select” some instructions for issue
  - Arbiter picks a subset of ready instructions for execution
  - Example policies: random, lower-first, oldest-first, critical-first
- Instructions read out from issue window and sent to execution

EXECUTE STAGE

- Read operands from physical register file and/or bypass network from other functional units
- Execute on functional unit
- Write result value to physical register file (or store buffer if store)
- Produce exception status, write to reorder buffer
- Free slot in instruction window

COMMIT STAGE

- Read completed instructions in-order from reorder buffer
  - (may need to wait for next oldest instruction to complete)
- If exception raised
  - flush pipeline, jump to exception handler
- Otherwise, release resources:
  - Free physical register used by last writer to same architectural register
  - Free reorder buffer slot
  - Free memory reorder buffer slot

ACKNOWLEDGEMENTS

- These slides contain material developed and copyright by:
  - Arvind (MIT)
  - Krste Asanovic (MIT/UCB)
  - Joel Emer (Intel/MIT)
  - James Hoe (CMU)
  - John Kubiatowicz (UCB)
  - David Patterson (UCB)

- MIT material derived from course 6.823
- UCB material derived from course CS252