Speculative Execution

Main Idea

- General: Execute instructions that may not be needed, before it is known whether or not they will be needed.
- Optimistic: Execute instructions whose correctness is uncertain.
- If execution turns out to be incorrect or unnecessary, discard or undo their results.
- Examples:
  - Exceptions
  - Branch prediction
  - Memory reordering (hoisting)

Speculative Execution for Branches

- Use prediction to guess branch direction.
- Execute along speculative path.
- Architectural state is not committed until after branch direction is resolved.
- Major advancement over delayed branch and squashing

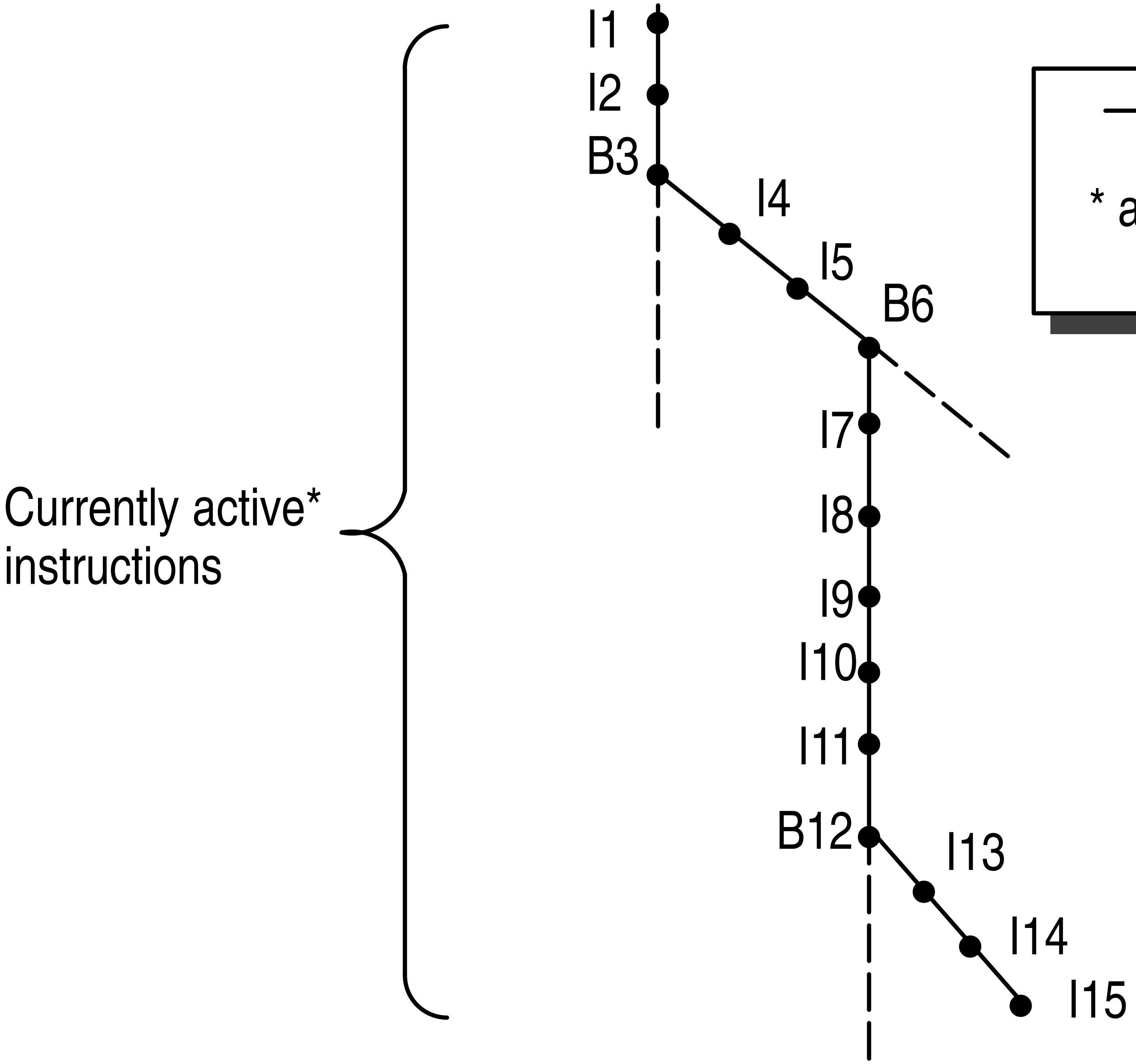
Degrees of Speculation

- Just Fetch?
- Fetch and Decode?
- Fetch, Decode, and Dispatch?
- Fetch, Decode, Dispatch, Complete with forwarding, but don't update architectural state. [Most common solution]

Levels of Speculation

- Typically, a CPU can speculate through multiple predicted but unresolved branches.
- The number of outstanding branches is finite.

- Speculative execution along predicted branch direction:



— :predicted control flow path

* active = dispatched, in execution, waiting execution, waiting retirement

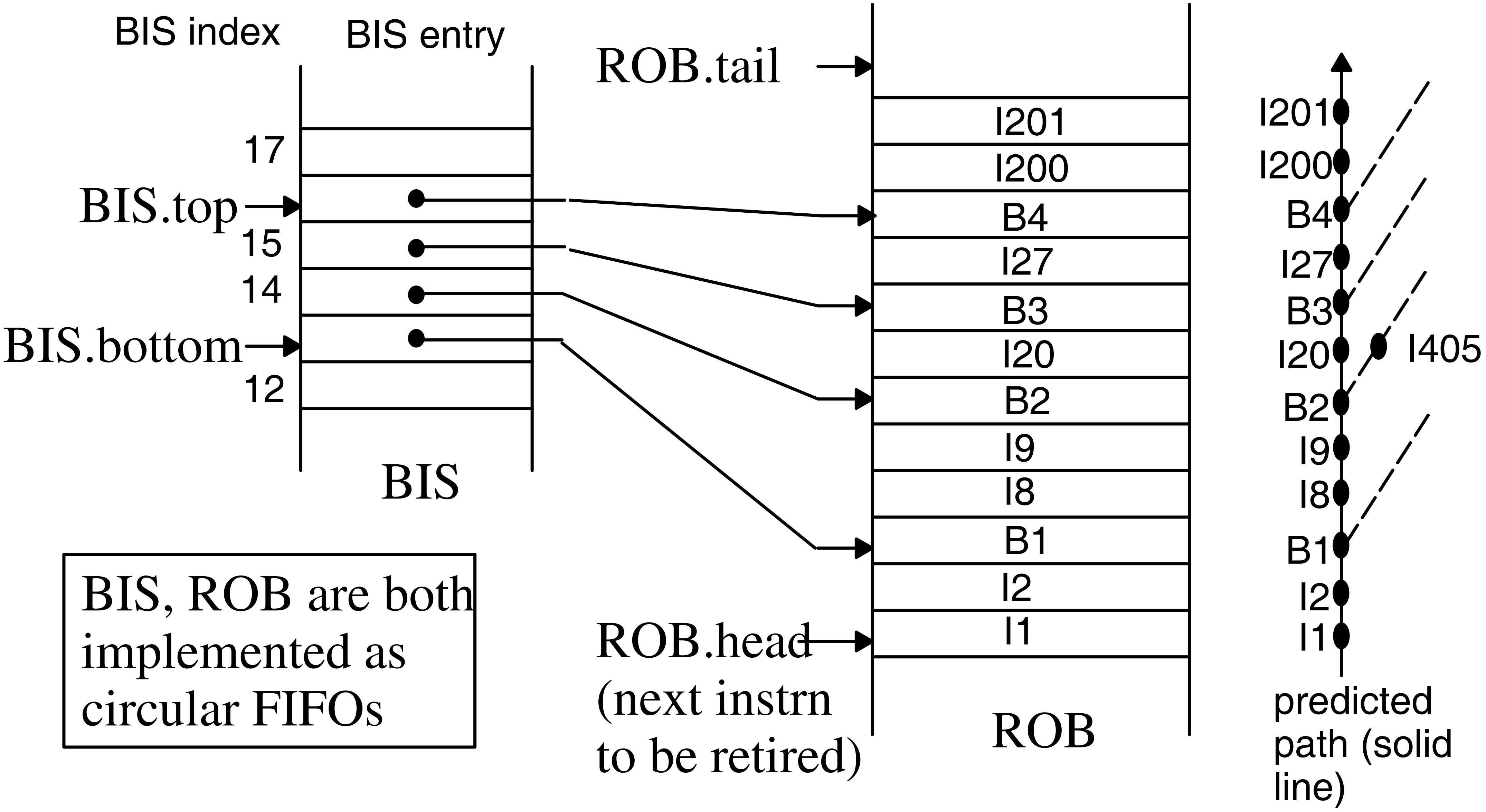
Here, instructions are speculatively executed along the predicted path following three unresolved branch instructions (B3, B6 and B12)

Level of speculation is 3

Handling Speculative Execution with a ROB

- Additional facilities needed:

► A branch instruction stack, BIS, implemented as a circular FIFO queue, with pointers for the stack top and stack bottom. Both pointers move, as seen below:



BIS, ROB are both implemented as circular FIFOs

Branch Tags

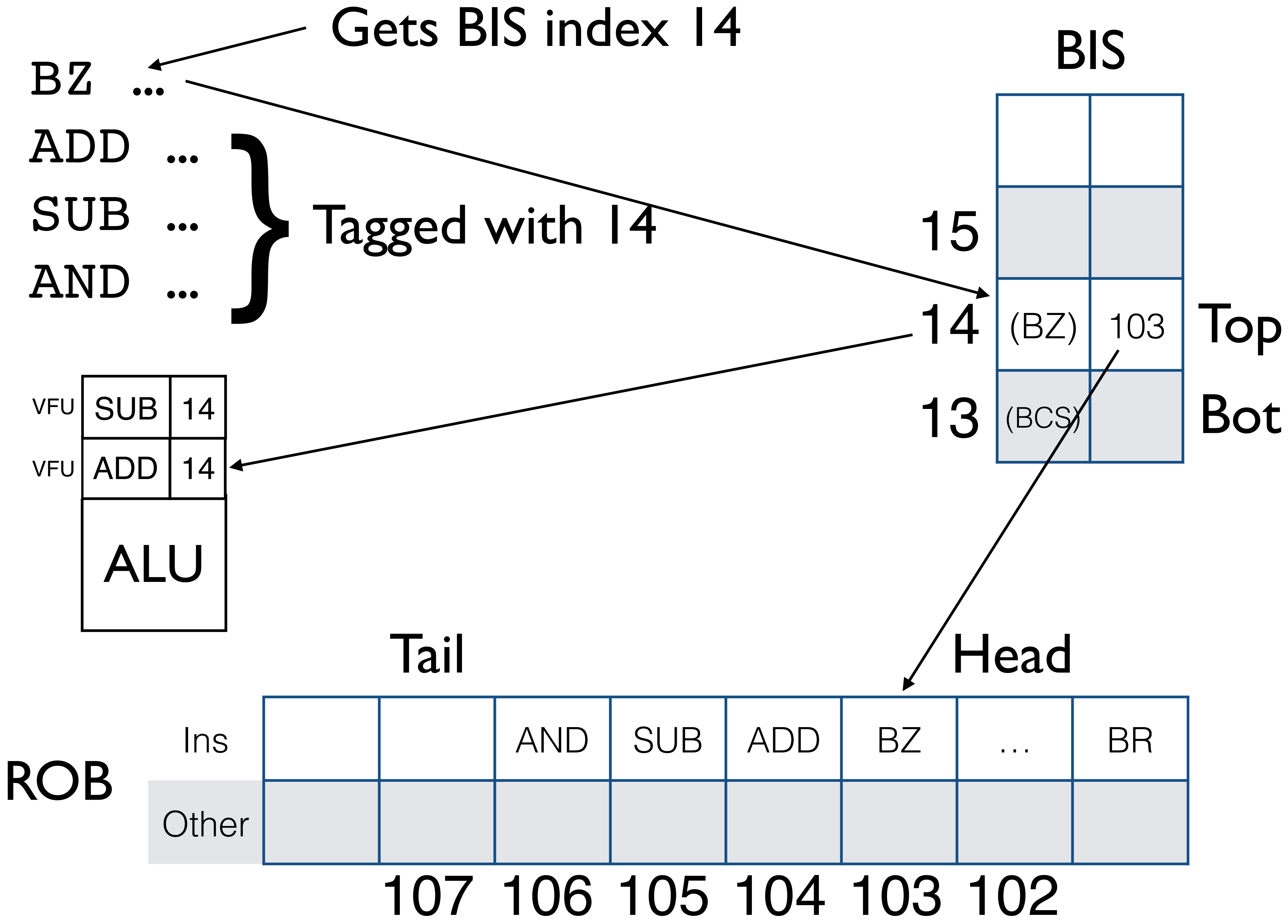
- Dispatched instructions are tagged with the BIS index of the closest preceding branch.
- Tag follows execution through to the RoB.
- Flush speculated instructions from RoB based on tag.

Dispatching Branches

- Allocate entries in: PRF, IQ, RoB, and BIS
- To allocate in BIS if not full:
    - Increment BIS.top to point to next free entry
    - BIS[top].rob = RoB entry for branch ins.

Dispatching Non-branch Instructions

- Allocate entries in: PRF, IQ, RoB
- Tag instructions in IQ with BIS.top



BZ ...

ADD ...

SUB ...

AND ...

Gets BIS index 14

Tagged with 14

BIS

15			
14	(BZ)	103	Top
13	(BCS)		Bot

VFU	SUB	14
VFU	ADD	14
	ALU	

ROB

		Tail				Head		
Ins			AND	SUB	ADD	BZ	…	BR
Other								
		107	106	105	104	103	102	

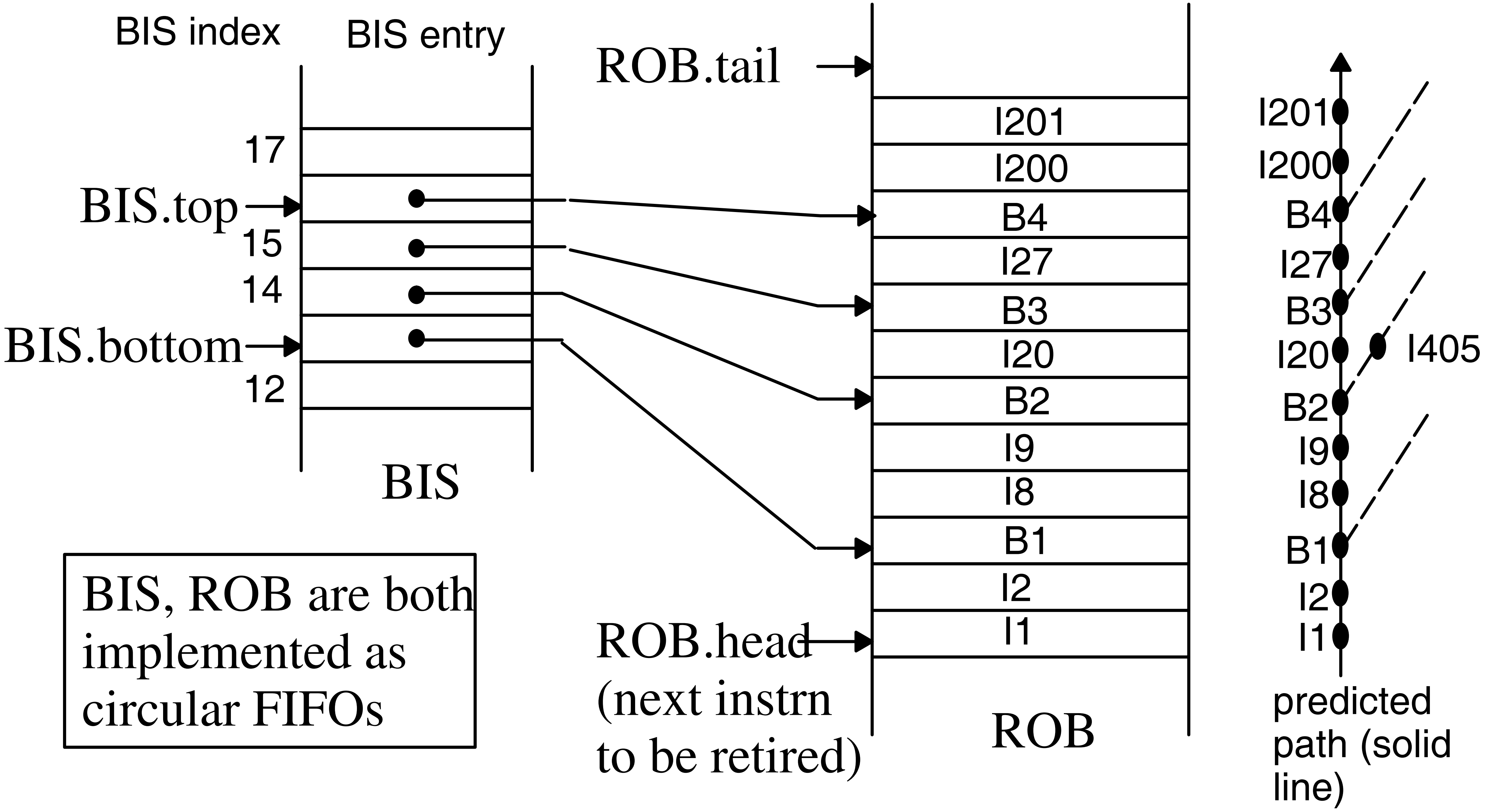
Handle Correctly-predicted Branch

- No special action on resolving branch.
- When committing RoB entry, increment BIS.bottom to free BIS entry.

Handle Mispredicted Branches

- Use BIS entry of mispredicted branch to locate branch in RoB.
- Flush from RoB all instructions following this branch.
  - Point RoB.tail to entry following branch.
- Flush from IQ and EX by broadcasting tags.
  - Broadcast branch tags from BIS
  - Decrement BIS.top until BIS.top points to mispredicted branch.

Example



BIS, ROB are both implemented as circular FIFOs

Example Facts

- Flushes everything following mispredicted branch in program order
- Much in common with exception handling with precise interrupts
- Needn’t wait for retirement to undo mispredicted branch.

Register Renaming & Speculation

- Speculated instructions allocated physical registers: Deallocate them
- Restore rename table to time of branch

Deallocating physical registers

- Integrated RoB and PRF:
  - Automatic with moving ROB.tail
- Separate RoB and PRF:
  - A: Walk RoB in reverse, freeing registers
  - B: Take checkpoint for each branch

Objective

- Situation:
  - Discover that branch in the **middle** of the instruction window was mispredicted.
- Objective:
  - Immediately reset front end of processor to time of mis-speculated instruction.
  - Only erase from RoB and IQ instructions **after** mispeculated one.

Restoring Rename Table

- Integrated RoB and PRF with future RAT and retirement RAT:
  - On discovering mispredict, stop dispatch
  - **Wait** for instructions prior to branch to commit, establishing precise architectural state at time of branch
  - Copy retirement RAT to future RAT
  - ▶ Treats mispredict like exception
  - ▶ Can be very slow

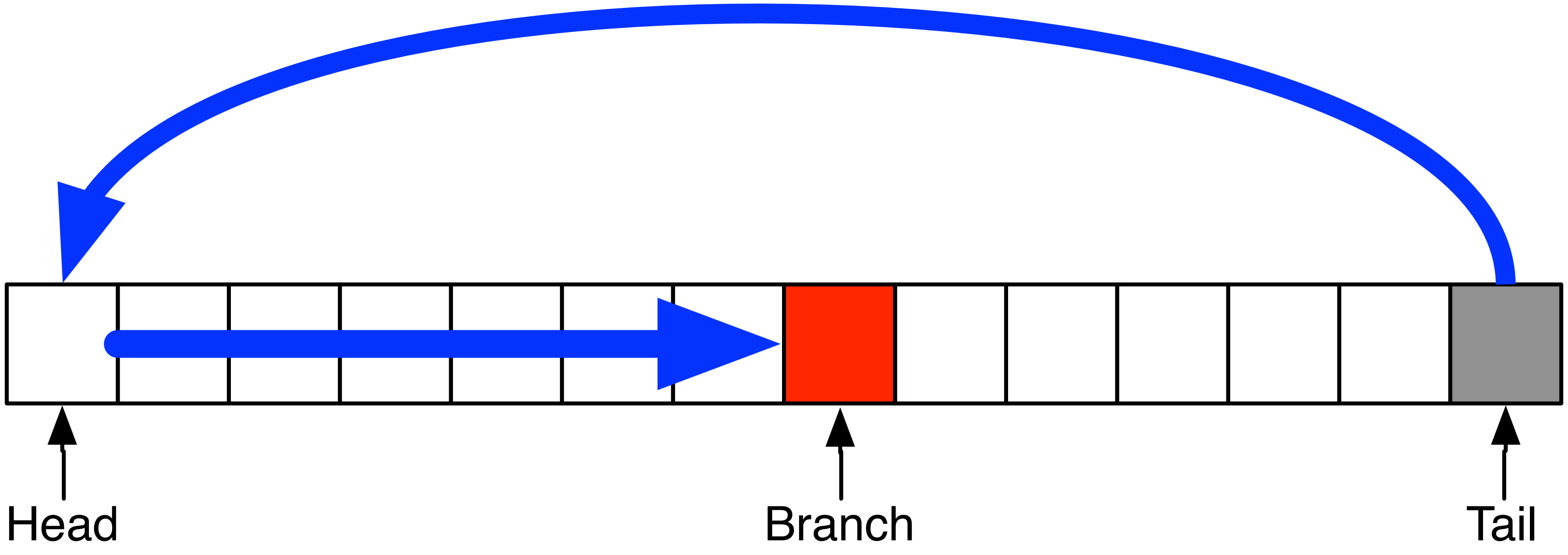
Restoring Rename Table Using Checkpoints

- Checkpoint RAT at time of branch
- Point BIS entry at checkpoint
- On mispredict, restore RAT from checkpoint

▸ Requires a lot of storage

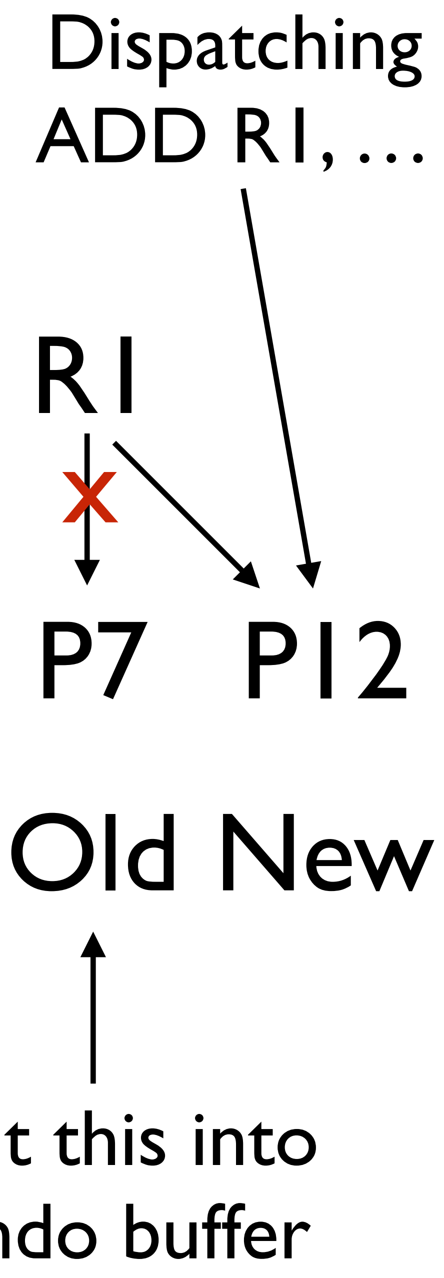
Restoring Rename Table "Walking Back"

- On mispredict:
  - Copy retirement RAT to future RAT
  - Fast forward from RoB.head to branch, updating RAT with arch to phys mappings found in RoB
  - "Walking backwards" towards the tail



Restoring Rename Table “Walking Forward”

- Keep “undo” queue for RAT modifications
  - On dispatch, store old value of RAT[dest] in RAT history buffer.
- On mispredict:
  - Rewind ROB.tail back to branch, undoing all RAT changes made after branch
  - “Walking forward” towards ROB.head



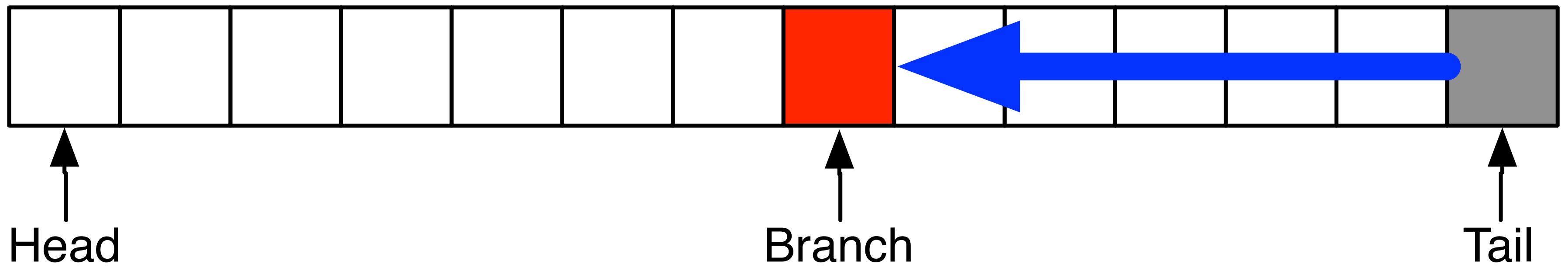
Dispatching ADD R1, …

R1

P7 P12

Old New

Put this into undo buffer



Example

```
ADD R1, …
MUL R0, …
AND R1, …
BZ …
ADDC R2, …
SUB R1, …
```

```
ADD P4, …
MUL P5, …
AND P6, …
BZ …
ADDC P7, …
SUB P8, …
```

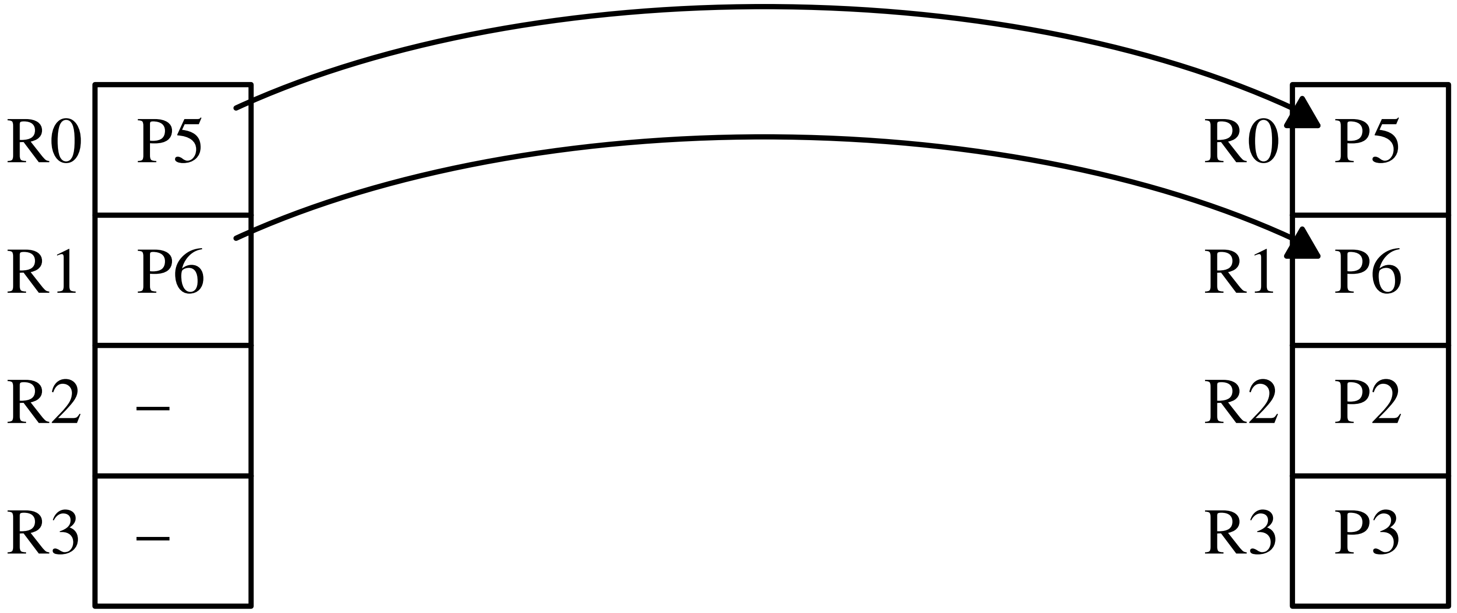
	Initially	ADD P4 (R1)	MUL P5 (R0)	AND P6 (R1)	BZ	ADDC P7 (R2)	SUB P8 (R1)
R0	P0	P0	P5	P5	P5	P5	P5
R1	P1	P4	P4	P6	P6	P6	P8
R2	P2	P2	P2	P2	P2	P7	P7
R3	P3	P3	P3	P3	P3	P3	P3

	Tail						Head	
Ins		SUB	ADDC	BZ	AND	MUL	ADD	
Arch		R1	R2		R1	R0	R1	
Phys		P8	P7		P6	P5	P4	
Other								

- **Alternative 1:** Treat mispredicted branch as an exception. Restore rename table by copying valid entries from the retirement RAT to the front end RAT (FE–RAT) after all instructions ahead of BZ are committed:

Retirement RAT (R–RAT) after committing instructions preceding BZ:

Restored front–end RAT: valid entries from R–RAT copied to FE–RAT



R0	P5
R1	P6
R2	–
R3	–

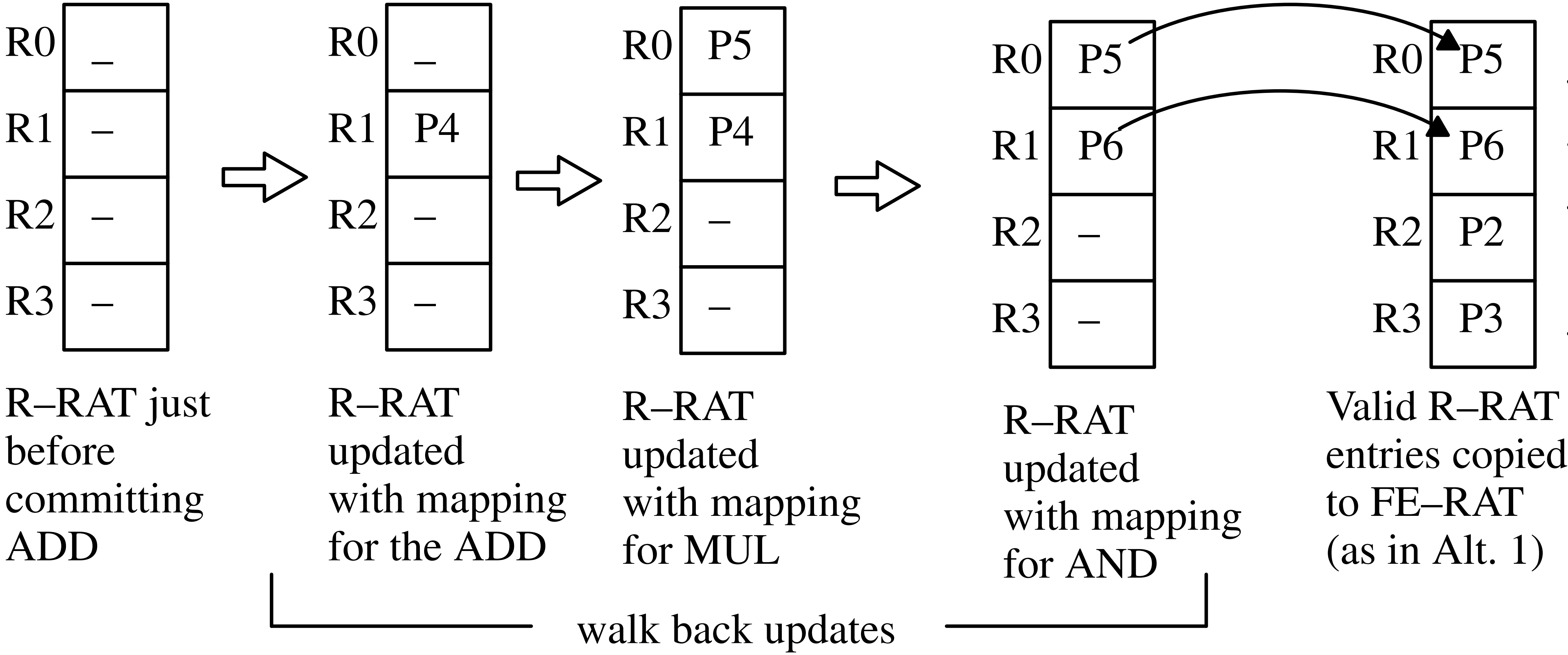
R0	P5
R1	P6
R2	P2
R3	P3

```
ADD R1, …
MUL R0, …
AND R1, …
BZ …
ADDC R2, …
SUB R1, …
```

R–RAT entries for R2 and R3 are not copied to the FE–RAT, as they ar not initialized

- **Alternative 2:** Checkpoint the rename table – works with all datapath variations – here rename table at the point of dispatching BZ is saved.

- **Alternative 3:** Improvement to Alternative 1 – use retirement RAT and **walk back** from the commit end of the ROB to the entry for the mispredicted branch:



R–RAT just before committing ADD

R–RAT updated with mapping for the ADD

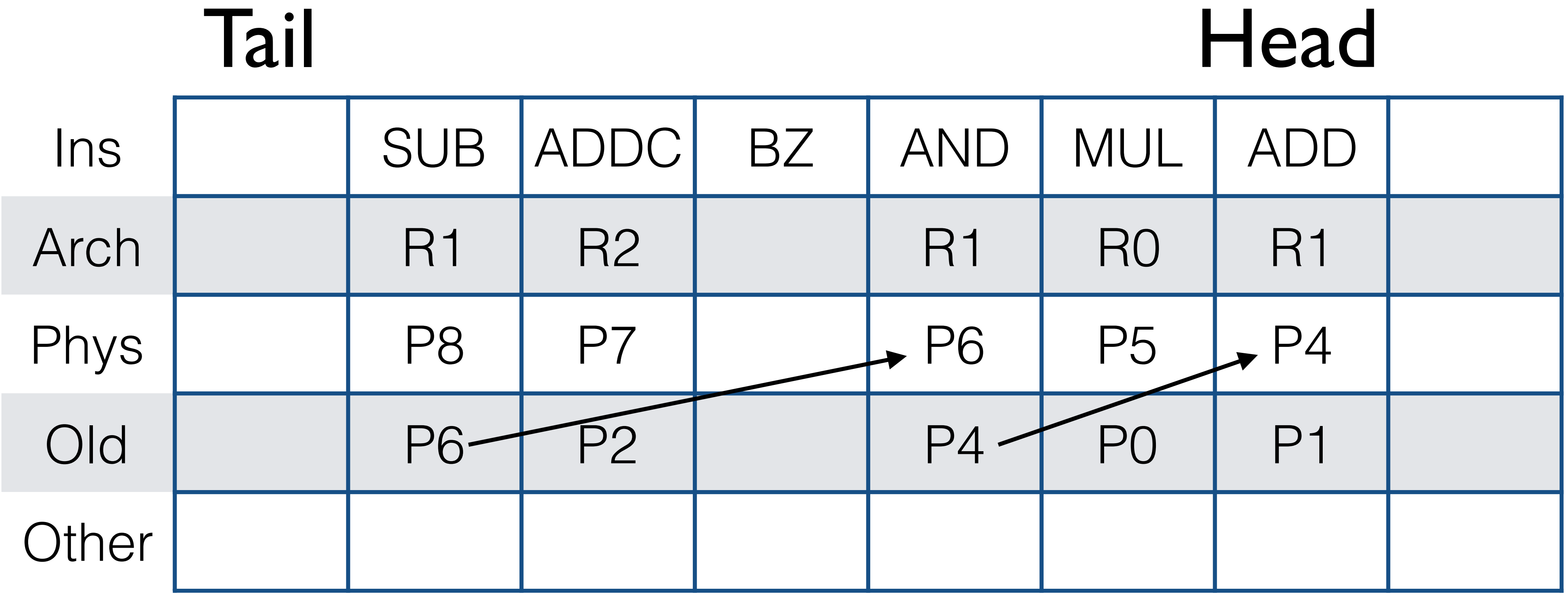
R–RAT updated with mapping for MUL

R–RAT updated with mapping for AND

Valid R–RAT entries copied to FE–RAT (as in Alt. 1)

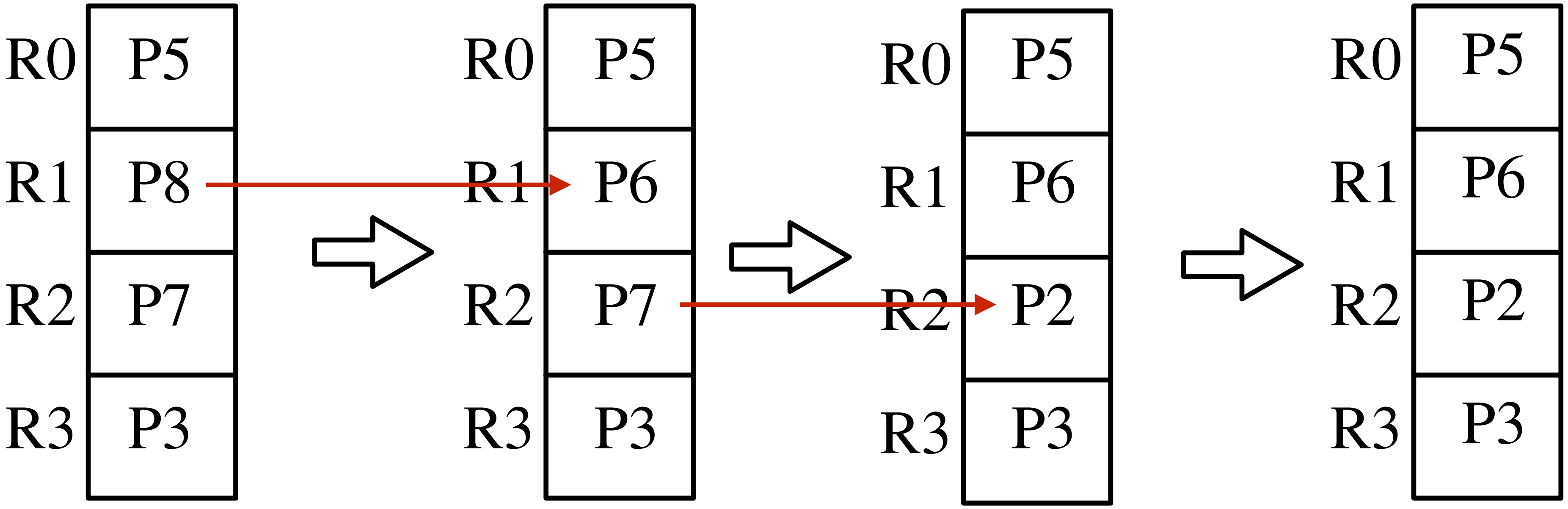
```
ADD R1,
MUL R0,
AND R1,
BZ …
ADDC R2
SUB R1,
ADD R1, …
MUL R0, …
AND R1, …
BZ …
ADDC R2, …
SUB R1, …
```

Alternative 4:



	Tail						Head	
Ins		SUB	ADDC	BZ	AND	MUL	ADD	
Arch		R1	R2		R1	R0	R1	
Phys		P8	P7		P6	P5	P4	
Old		P6	P2		P4	P0	P1	
Other								

Rename table (RAT) updates on walking forward:



R0	P5
R1	P8
R2	P7
R3	P3

RAT just after dispatching SUB

R0	P5
R1	P6
R2	P7
R3	P3

RAT updated with old mapping overwritten by SUB

Free P8

R0	P5
R1	P6
R2	P2
R3	P3

RAT updated with old mapping overwritten by ADDC

Free P7

R0	P5
R1	P6
R2	P2
R3	P3

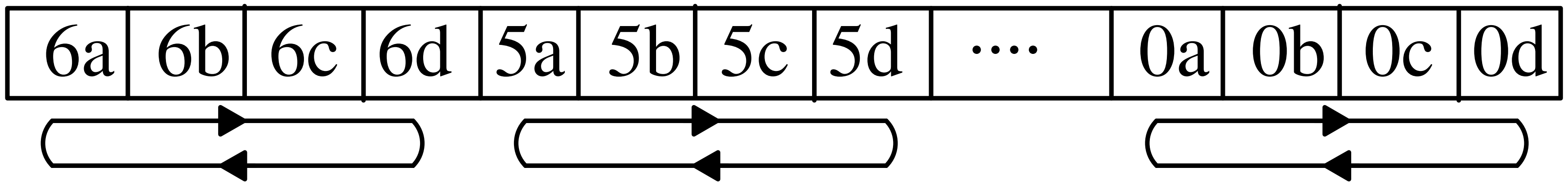
No updates for BZ: this is the restored RAT

walk forward updates

- Checkpointing is the fastest scheme – hardware implementation used should avoid entry–by–entry copying.

  Example: fast, parallel restoration similar to that used in the AMD K6:

  – One entry in the rename table: permits four checkpoints, “a” through “d”; each entry uses 7 bits (bits 6 through 0) to name a destination



6a	6b	6c	6d	5a	5b	5c	5d	····	0a	0b	0c	0d

  – For each bit, there are 4 values – bits “a” through “d”, corresponding to the 4 checkpoints. The bits holding these 4 values are set up logically as a circular shift register, as indicated by the arrows

Restoring Rename Table Analysis

- Walking techniques sequential and slow
  - Dependent on position of branch in RoB
- RAT restoration technique influences how physical registers are freed.
- Hardware requirements:
  - Most: Checkpointing
  - Walking Forward
  - Least: Walking Back or waiting