Assertion-Based Microarchitecture Design for Improved Fault Tolerance

Vimal K. Reddy

Ahmed S. Al-Zawawi, Eric Rotenberg

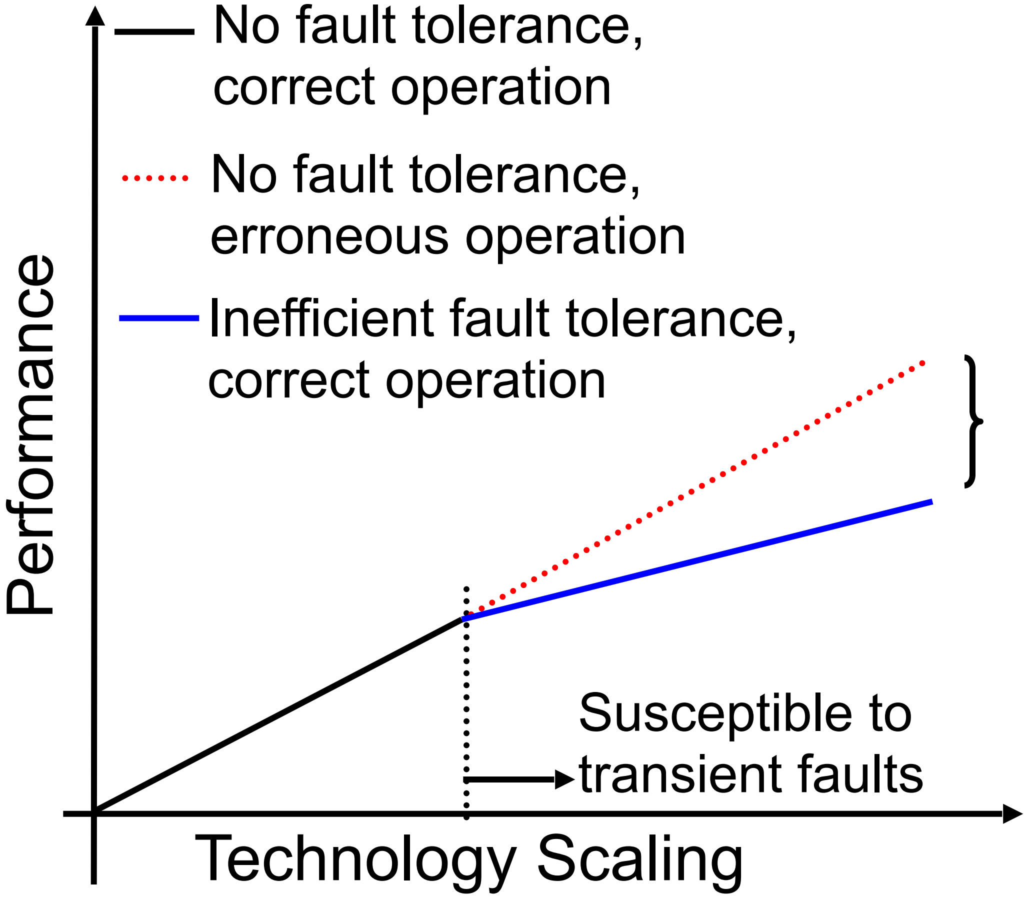
Center for Embedded Systems Research (CESR)
Department of Electrical & Computer Engineering
North Carolina State University

Motivation

- Technology scaling
  - Smaller, faster transistors
  - Transistors more susceptible to transient faults

- How to build reliable processors using unreliable transistors?

Motivation



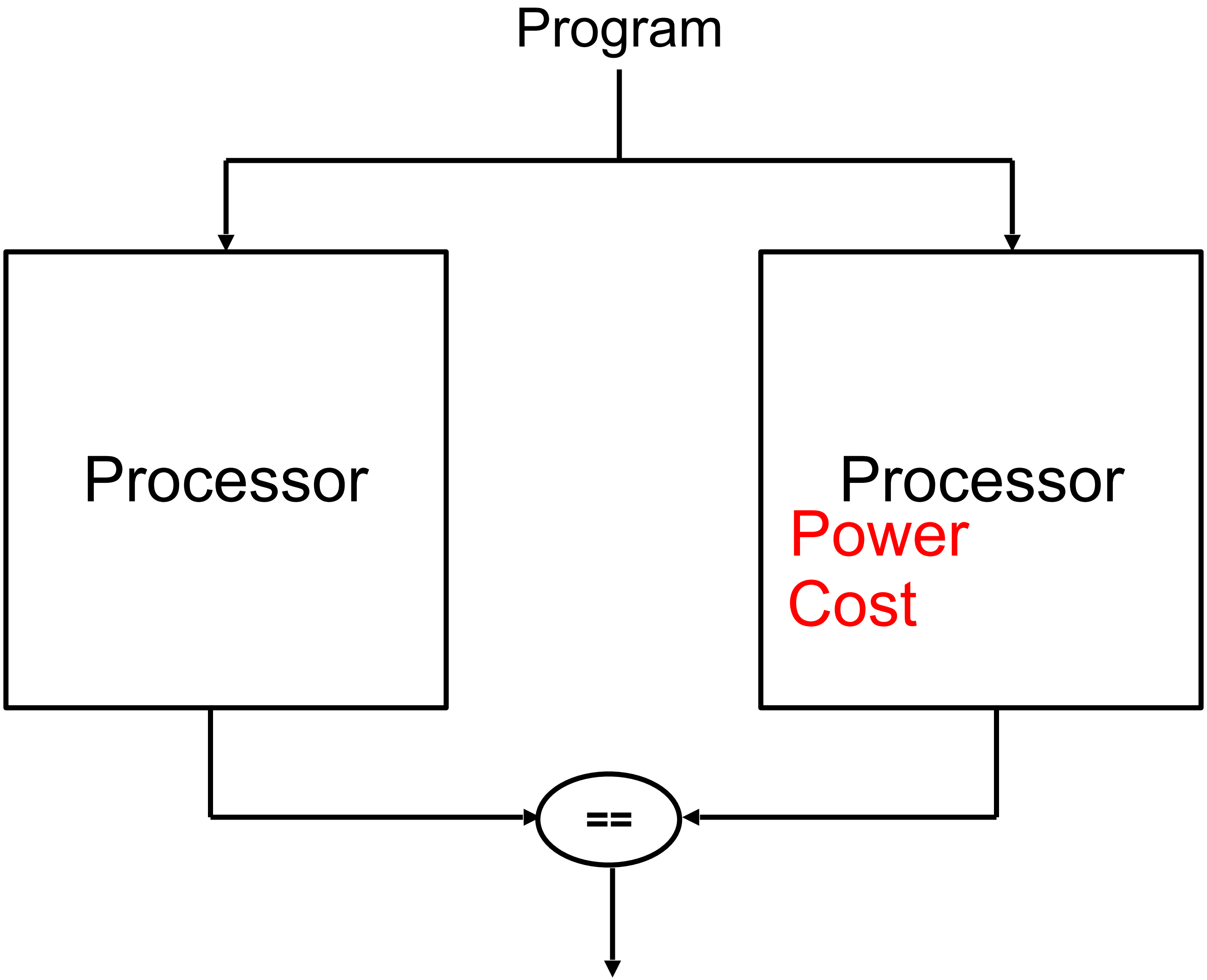
Due to high overheads of inefficient fault tolerance

Need efficient fault tolerance solutions

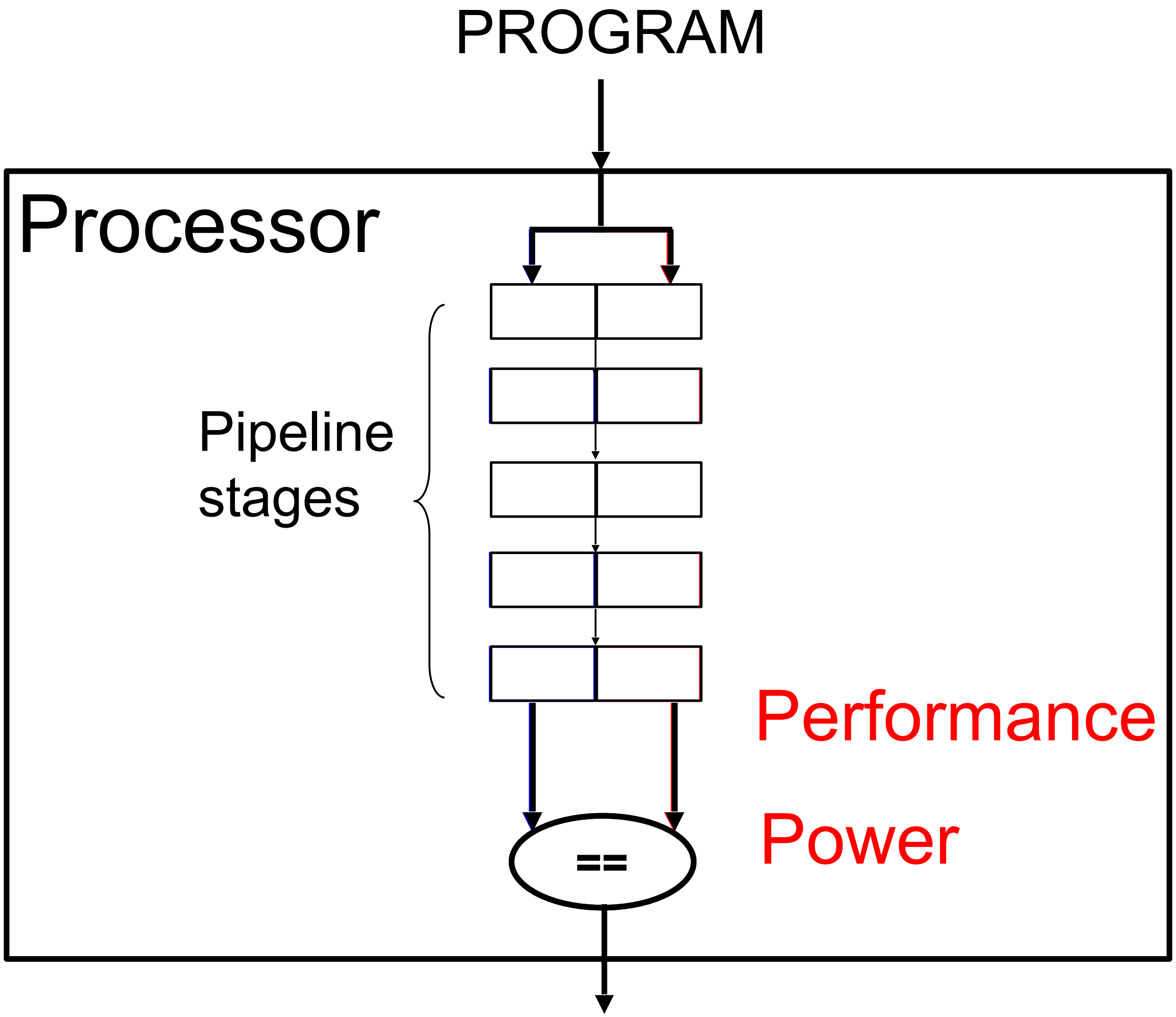
Redundant Multithreading (RMT)

- Duplicate a program and compare outcomes to detect transient faults
- Positives
  - Simple and straightforward
  - General solution
  - Complete fault coverage
- Negatives
  - High overhead

RMT using an extra processor



RMT using simultaneous multithreading



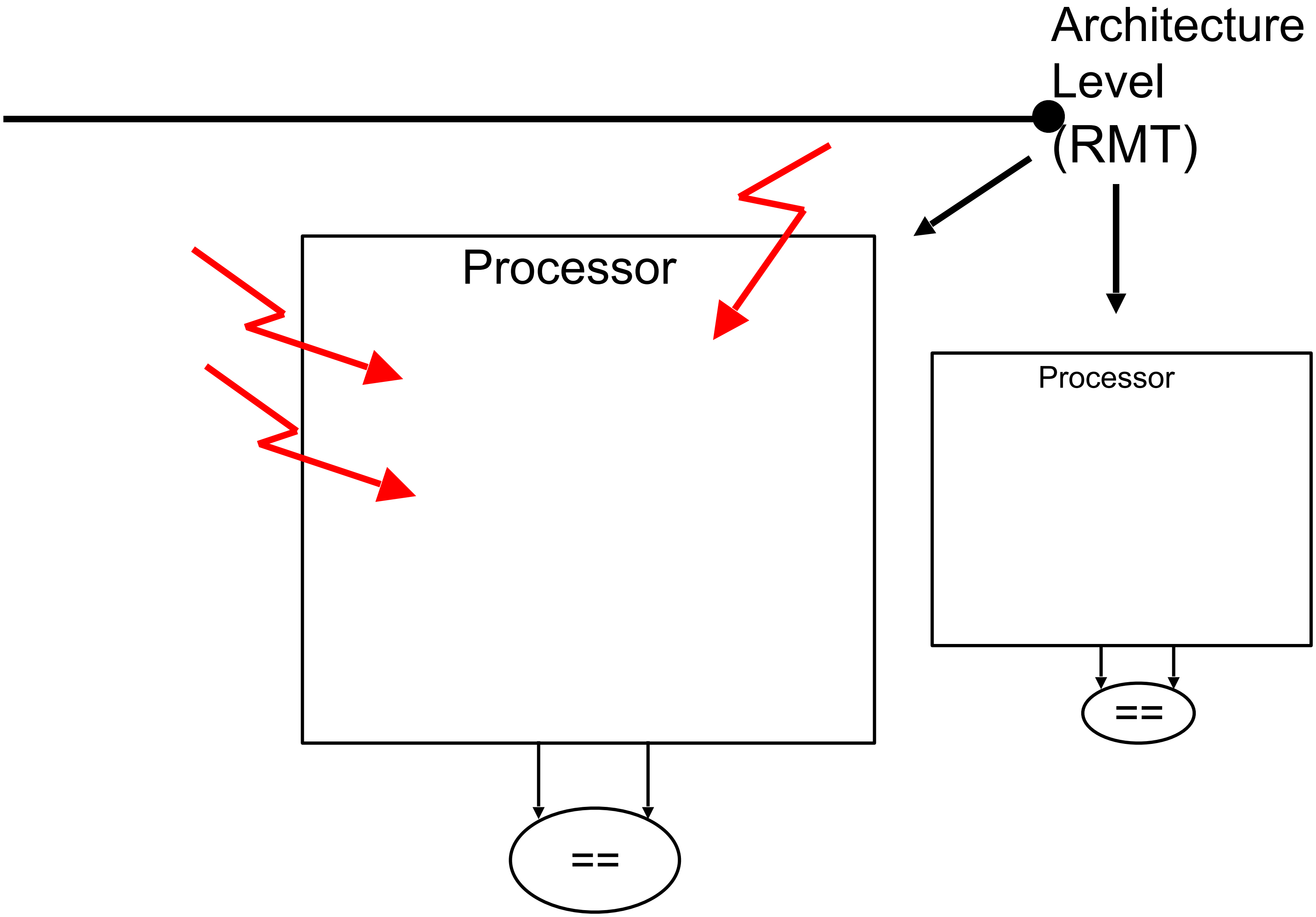
Alternate solution: Targeted fault checks

- Add regimen of fault checks
- Specific to logic block
    - Arbitrary latches: Robust latch design
    - Arbitrary gates: Self-checking logic
    - FSM: Self-checking FSM designs
    - ALU: Self-checking ALUs, RESO
    - Storage and buses: Parity, ECC
- Positives
    - No overhead of duplicating the program
- Negatives
    - Not general, i.e., many types of checks needed

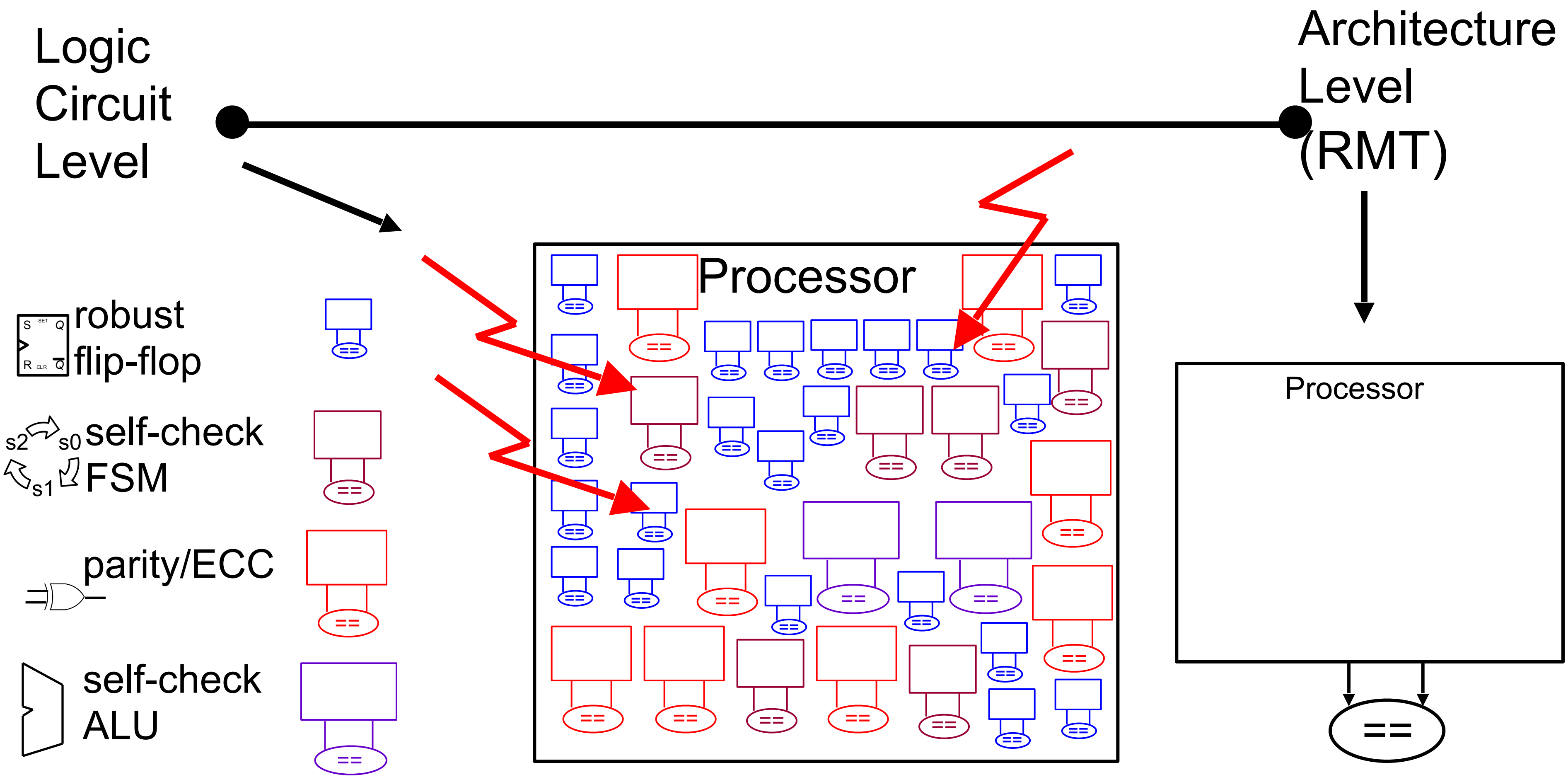
Our contribution: Microarchitectural Assertions

- Novel class of fault checks
- Key Idea: Confirm µarch. "truths" within processor pipeline
- Catch-all checks for microarchitecture mechanisms
- Positives
  - Broad coverage (catch-all checks)
  - Very low-overhead solution (no redundant execution)

Spectrum of fault checks



Spectrum of fault checks



Logic Circuit Level

Architecture Level (RMT)

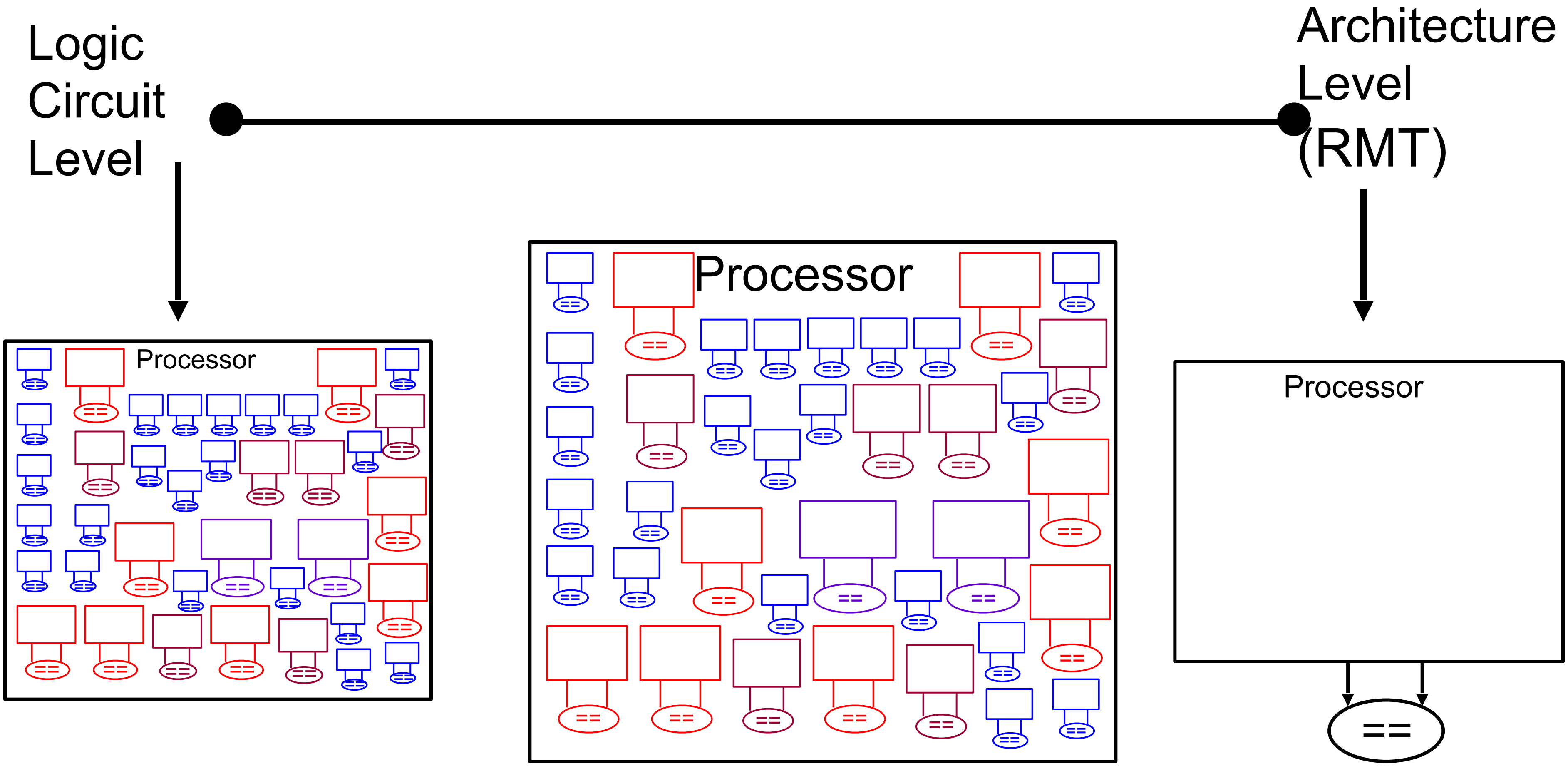
robust flip-flop

self-check FSM

parity/ECC

self-check ALU

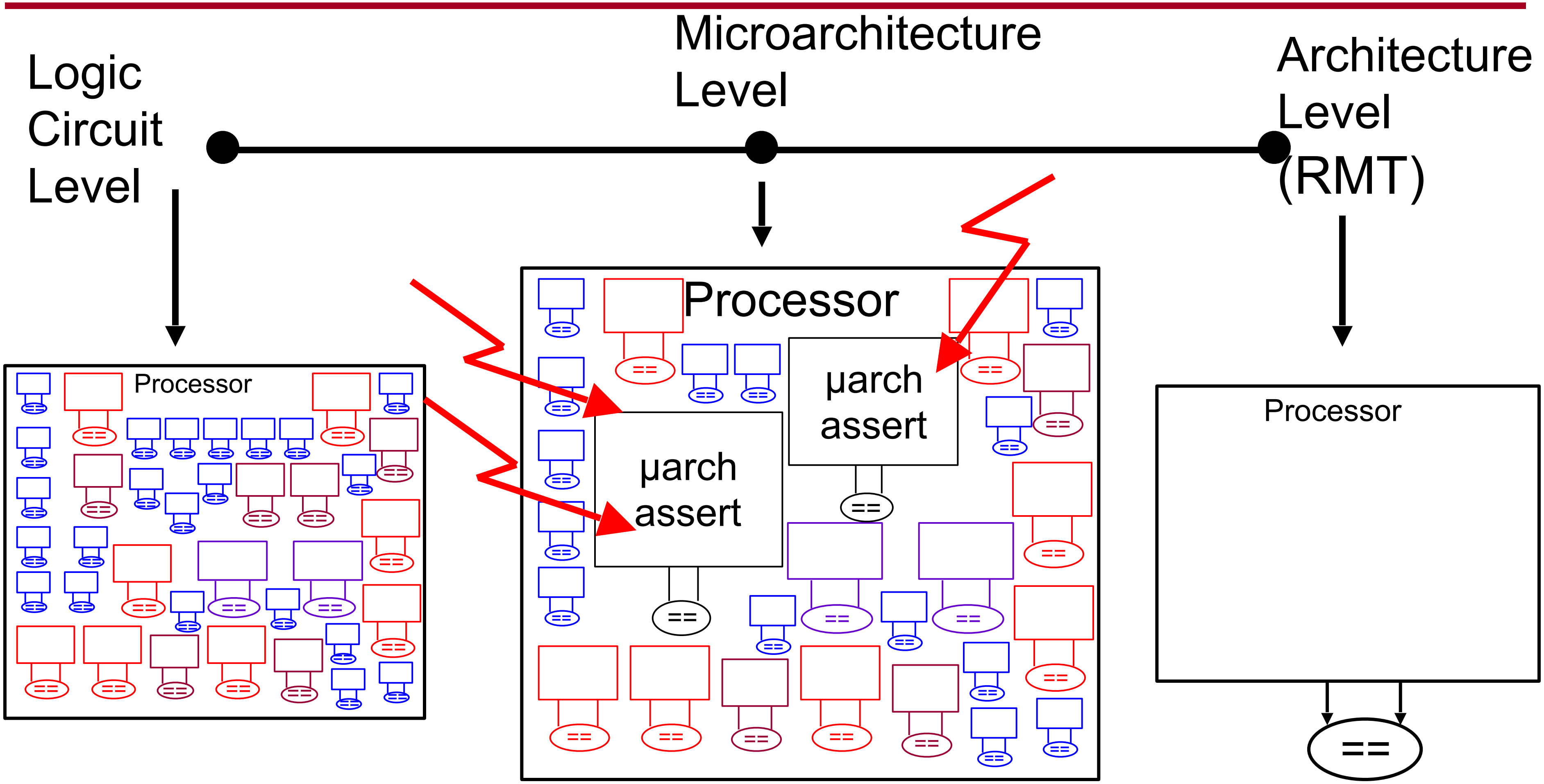
Spectrum of fault checks



Logic Circuit Level

Architecture Level (RMT)

Spectrum of fault checks



Logic Circuit Level

Microarchitecture Level

Architecture Level (RMT)

Examples of Microarchitectural Assertions

- Register Name Authentication (RNA)
  - Aims to detect faults in renaming unit
  - Asserts consistencies in renaming unit
    - Exploits inherent redundancy in renaming structures
    - Asserts expected physical register states

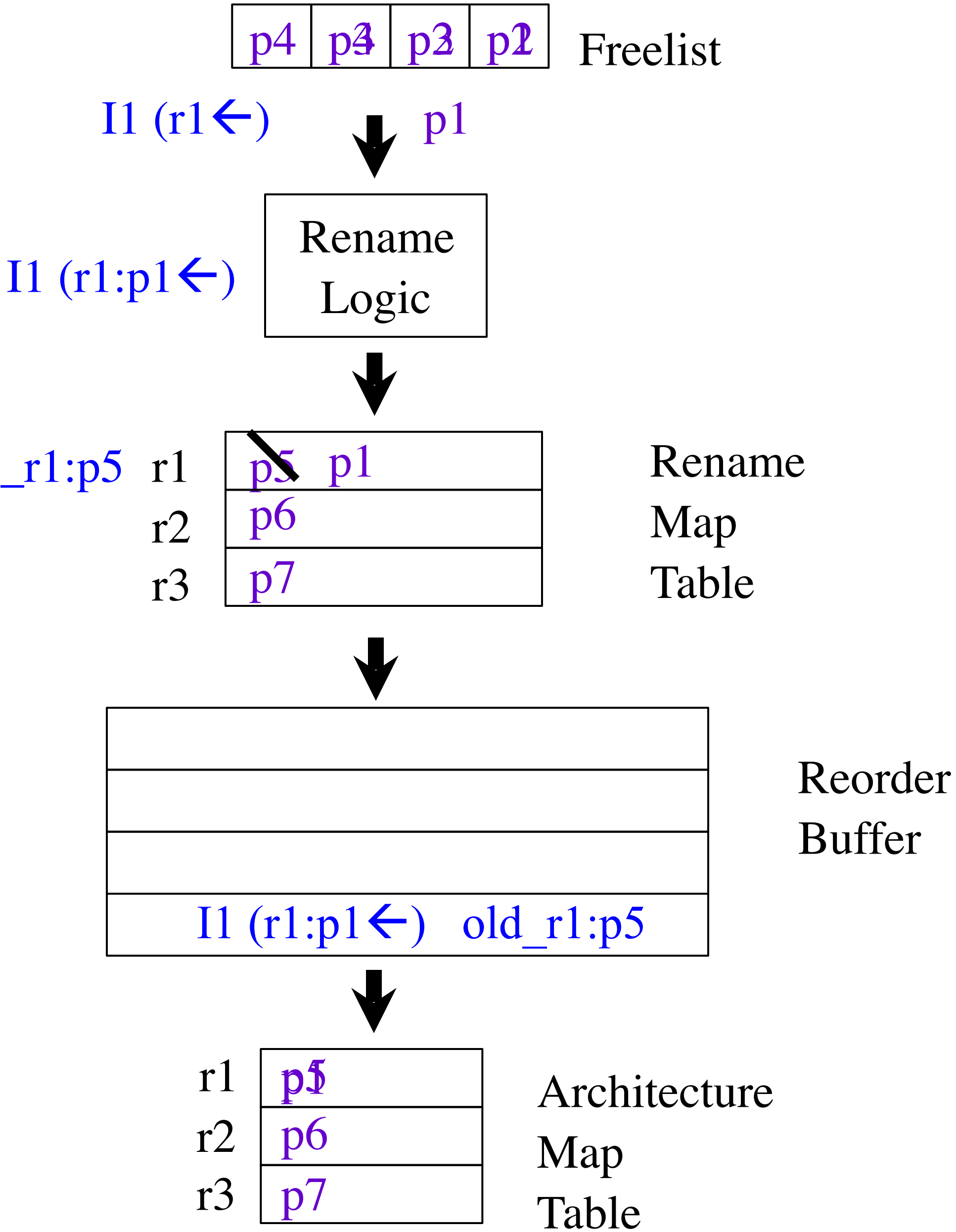
- Timestamp-based Assertion Checking (TAC)
  - Aims to detect faults in issue unit
  - Asserts sequential order among data dependent instructions
  - Uses timestamps

Program

I1 (r1←)

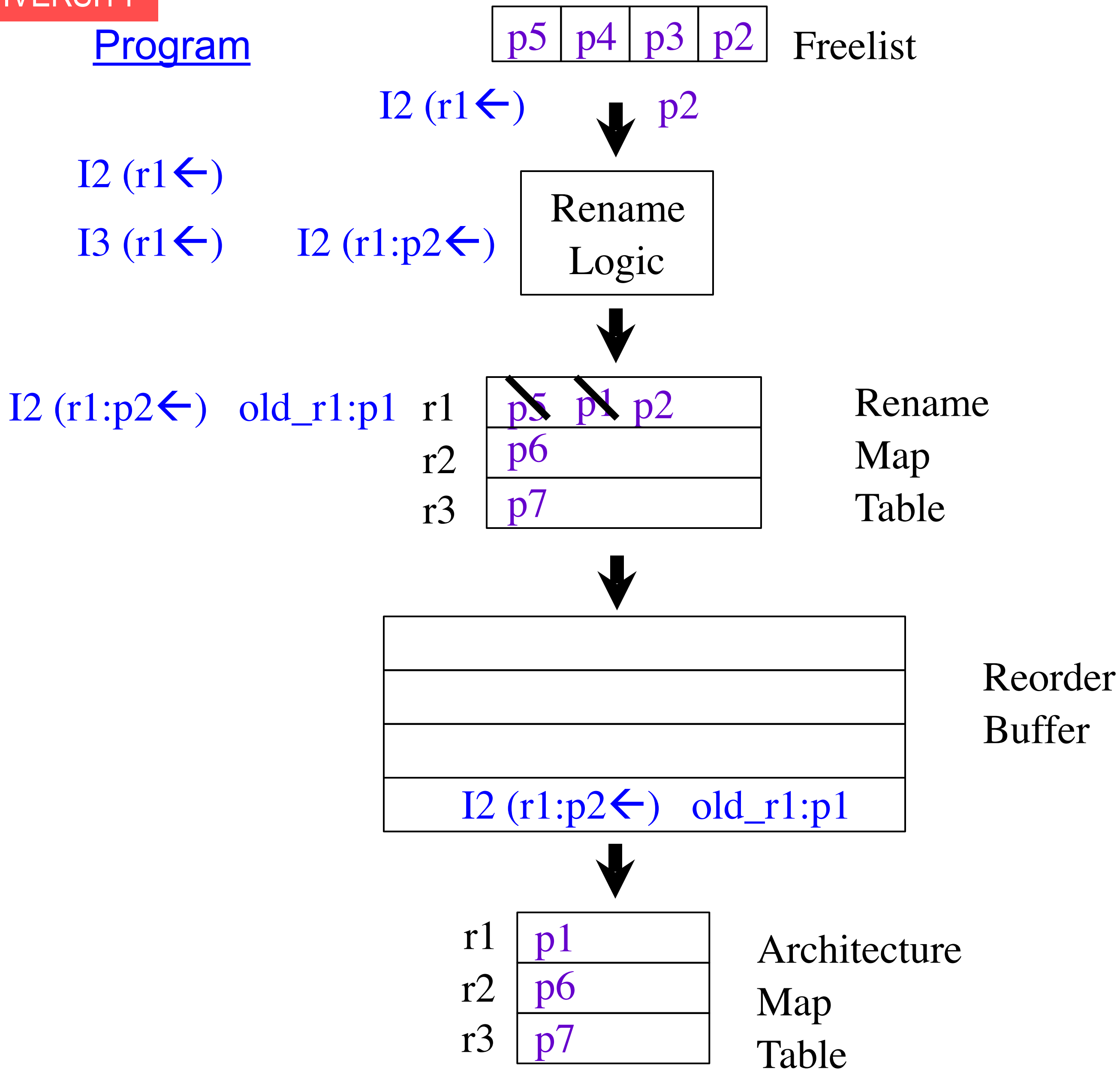
I2 (r1←)

I3 (r1←)



I1 (r1:p1←) old_r1:p5

I1 retires



Program

I2 (r1←)

I3 (r1←)

p5	p4	p3	p2

Freelist

I2 (r1←) p2

Rename Logic

I2 (r1:p2←)

I2 (r1:p2←) old_r1:p1

r1	~~p5~~ ~~p1~~ p2
r2	p6
r3	p7

Rename Map Table

Reorder Buffer

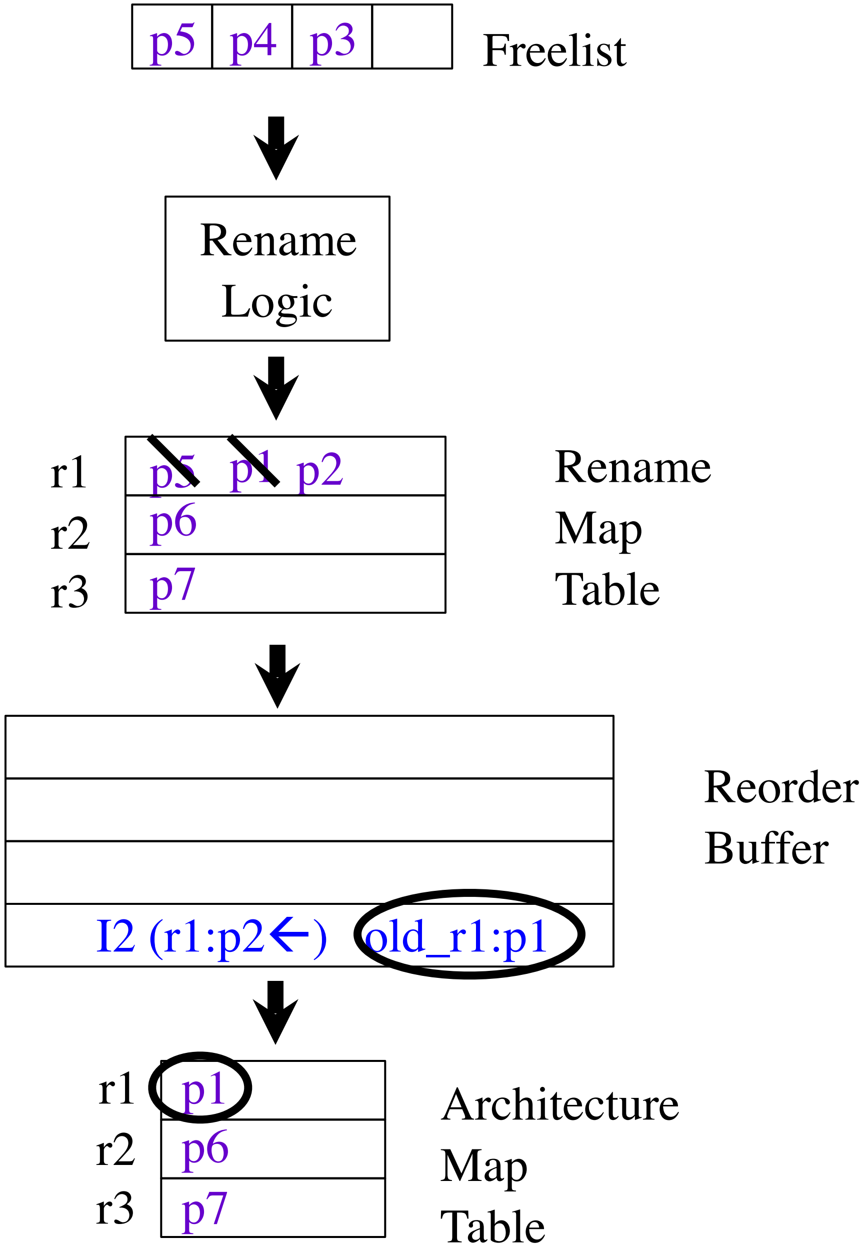
I2 (r1:p2←) old_r1:p1

r1	p1
r2	p6
r3	p7

Architecture Map Table

Program

I3 (r1←)



p5	p4	p3	

Freelist

Rename Logic

r1	~~p5~~ ~~p1~~ p2
r2	p6
r3	p7

Rename Map Table

Reorder Buffer

I2 (r1:p2←) old_r1:p1

r1	p1
r2	p6
r3	p7

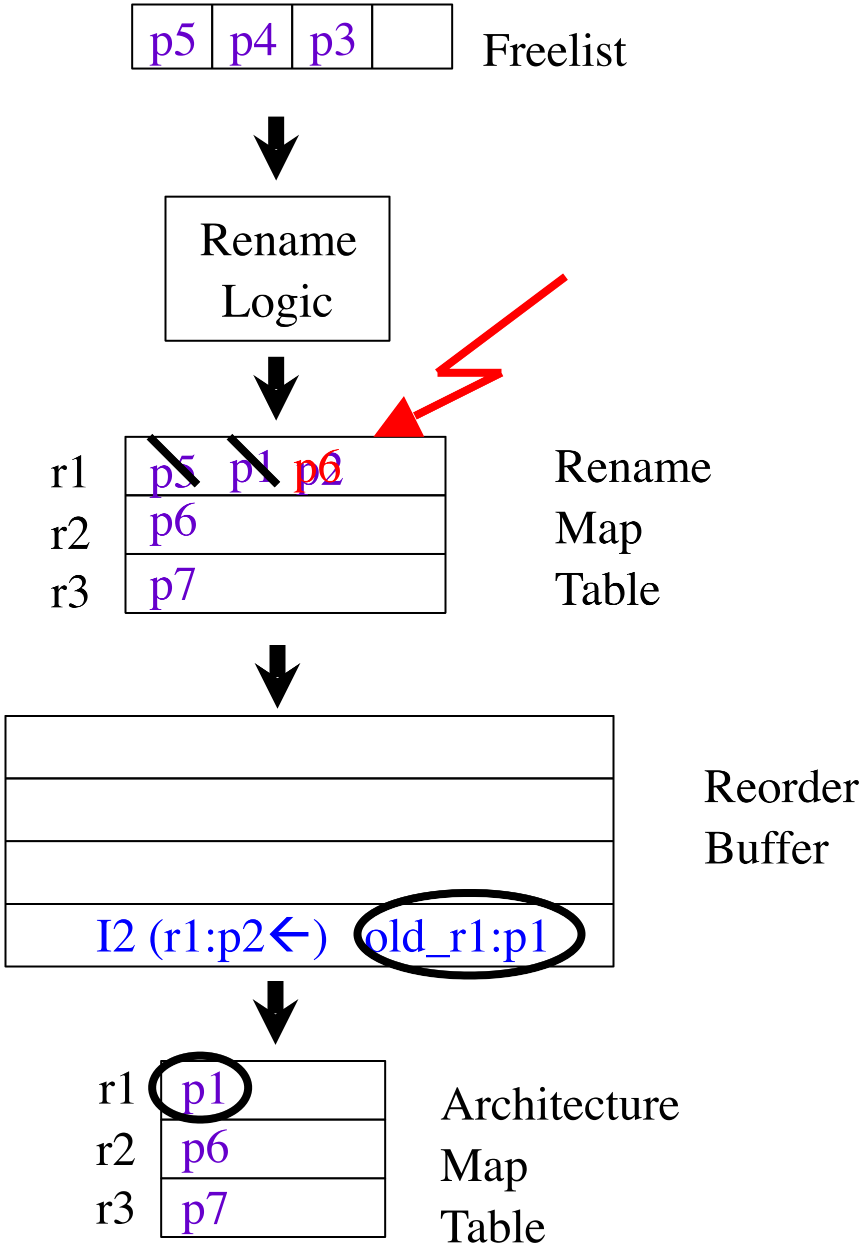
Architecture Map Table

Observation

r1's old mapping == r1's arch mapping

Program

I3 (r1←)



p5 p4 p3

Freelist

Rename Logic

r1	~~p5~~ ~~p1~~ p6 p2
r2	p6
r3	p7

Rename Map Table

Reorder Buffer

I2 (r1:p2←) old_r1:p1

r1	p1
r2	p6
r3	p7

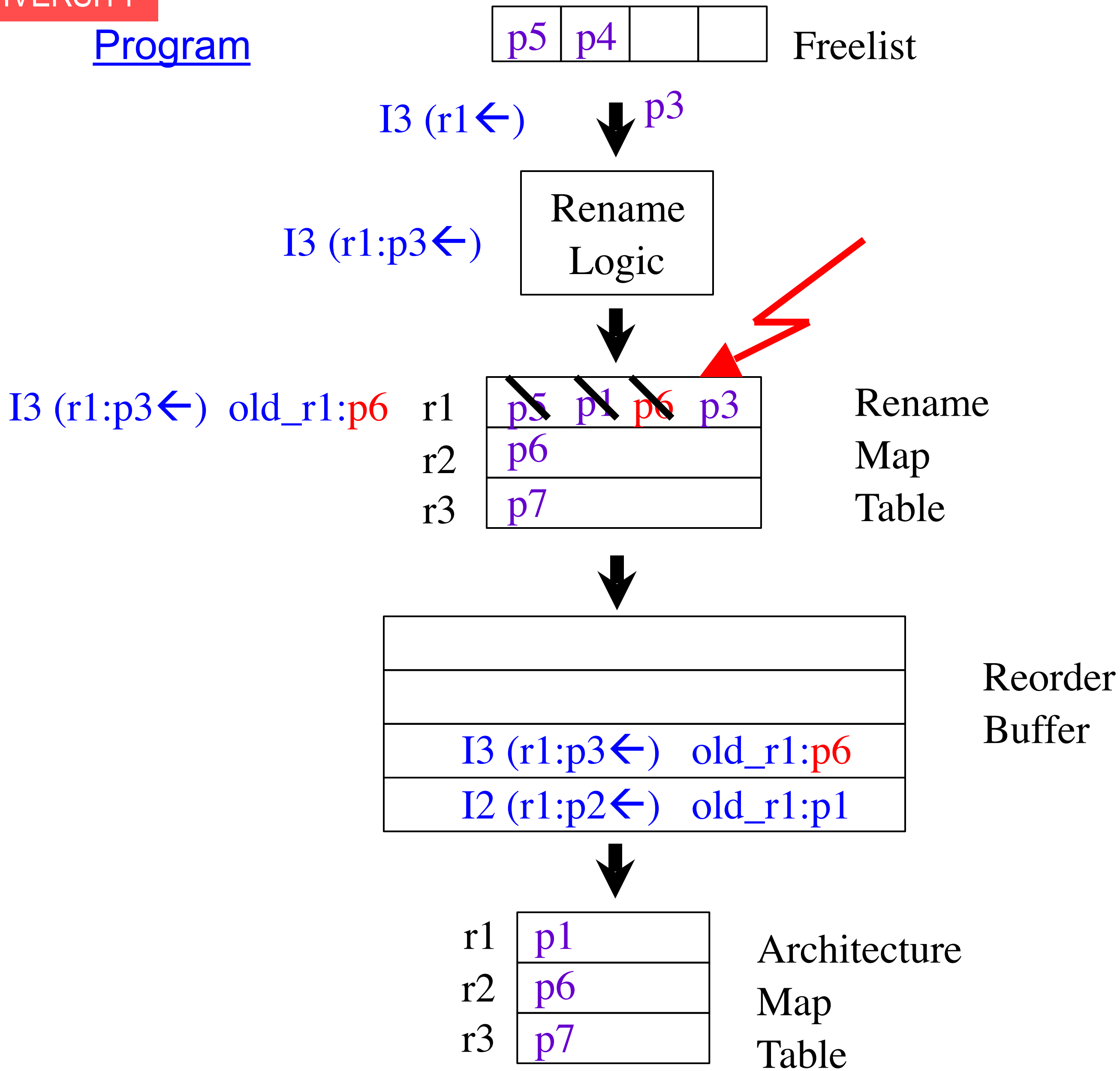
Architecture Map Table

RNA prev mapping check

old_r1 == arch_r1?

p1 == p1?

Program



p5	p4		

Freelist

I3 (r1←) p3

Rename Logic

I3 (r1:p3←)

I3 (r1:p3←) old_r1:p6

r1	~~p5~~ ~~p1~~ ~~p6~~ p3
r2	p6
r3	p7

Rename Map Table

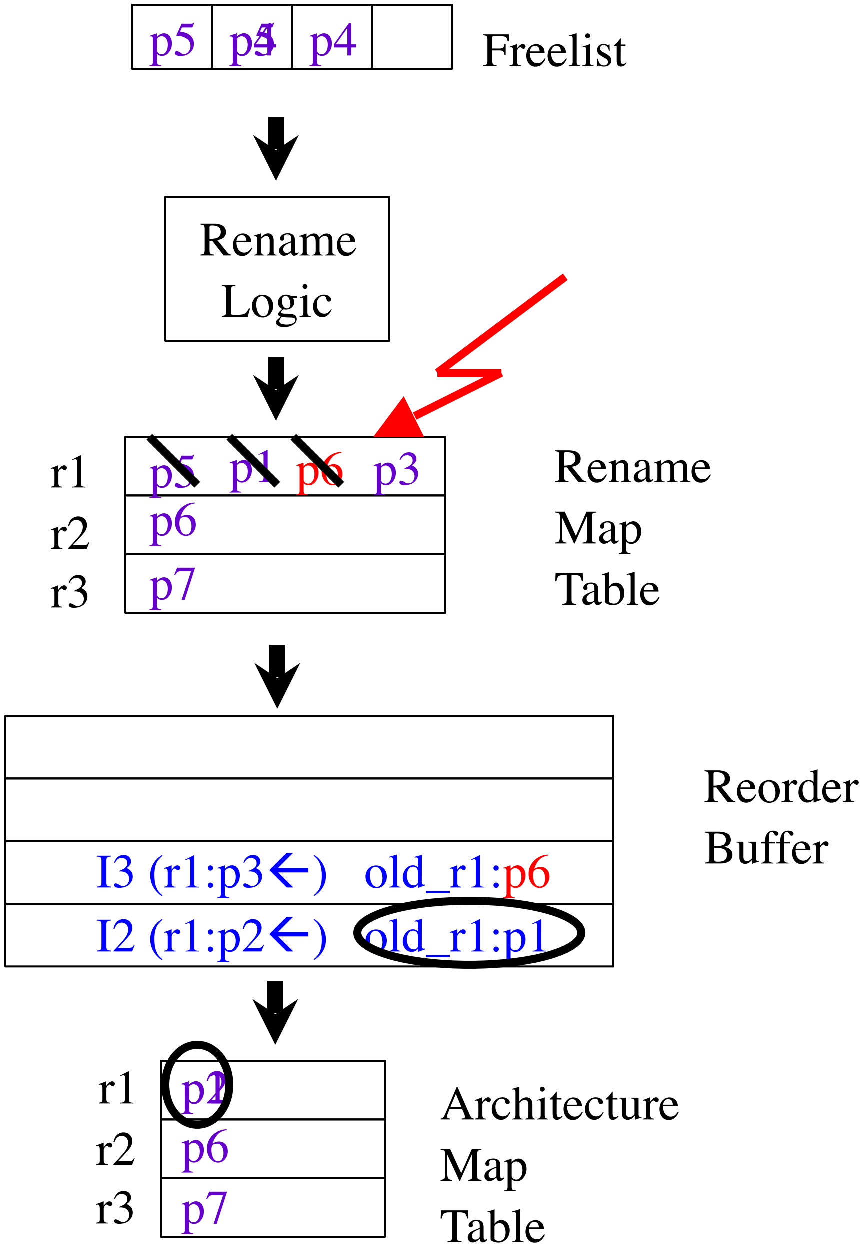
I3 (r1:p3←) old_r1:p6
I2 (r1:p2←) old_r1:p1

Reorder Buffer

r1	p1
r2	p6
r3	p7

Architecture Map Table

Program



p5 | p4 | p4 | Freelist

Rename Logic

r1 ~~p5~~ ~~p1~~ ~~p6~~ p3

r2 p6

r3 p7

Rename Map Table

I3 (r1:p3←) old_r1:p6

I2 (r1:p2←) old_r1:p1

Reorder Buffer

r1 p1

r2 p6

r3 p7

Architecture Map Table

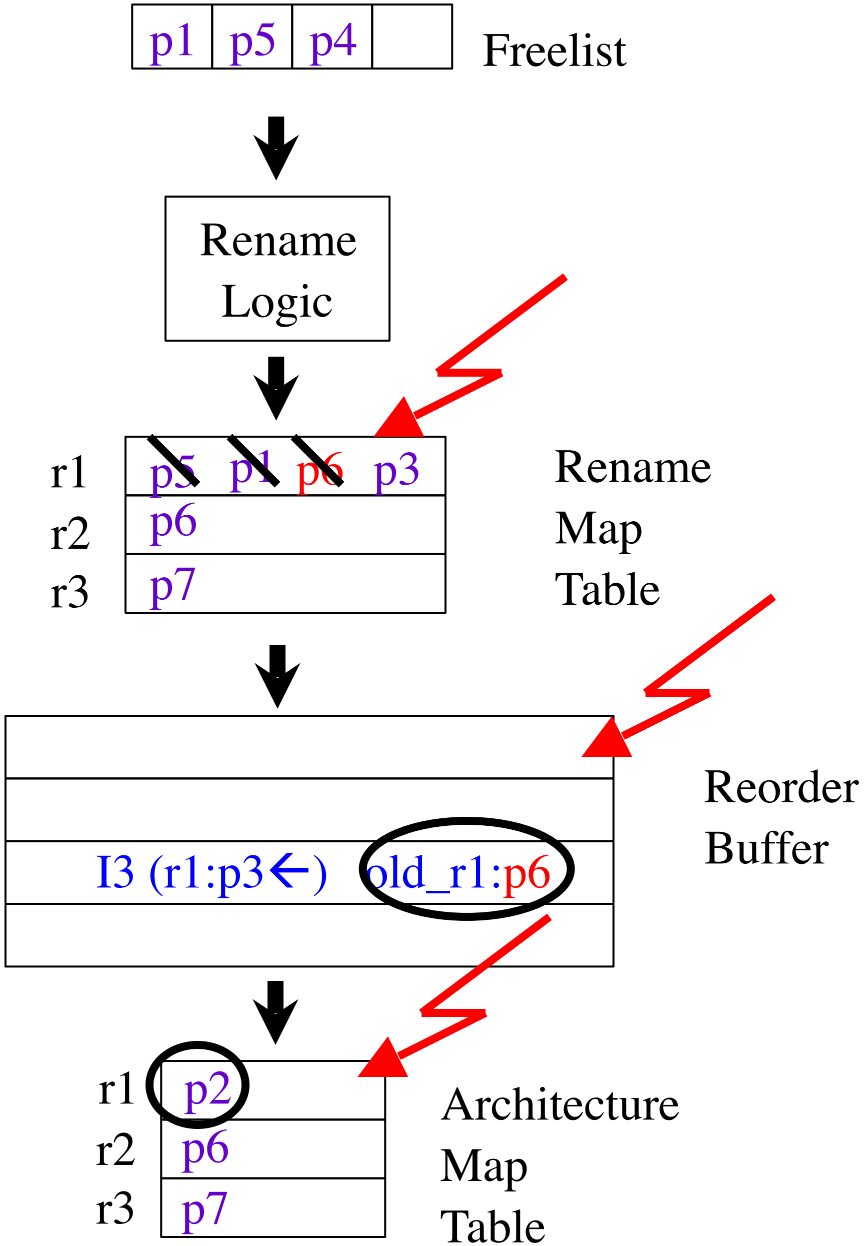
RNA prev mapping check

old_r1 == arch_r1?

p1 == p1?

I2 retires

Program



RNA prev mapping check

old_r1 == arch_r1?

p6 == p2?

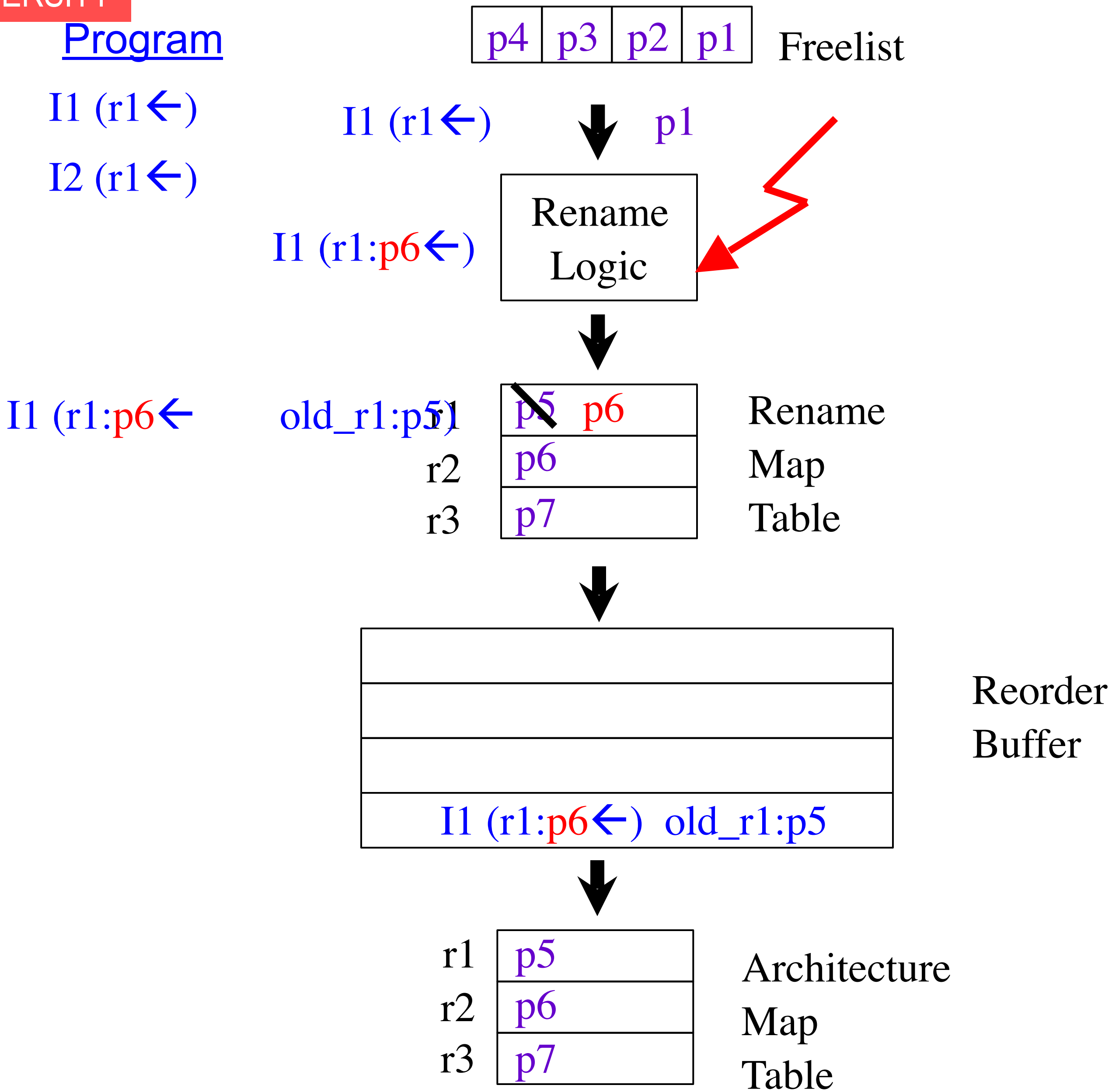
FAULT DETECTED

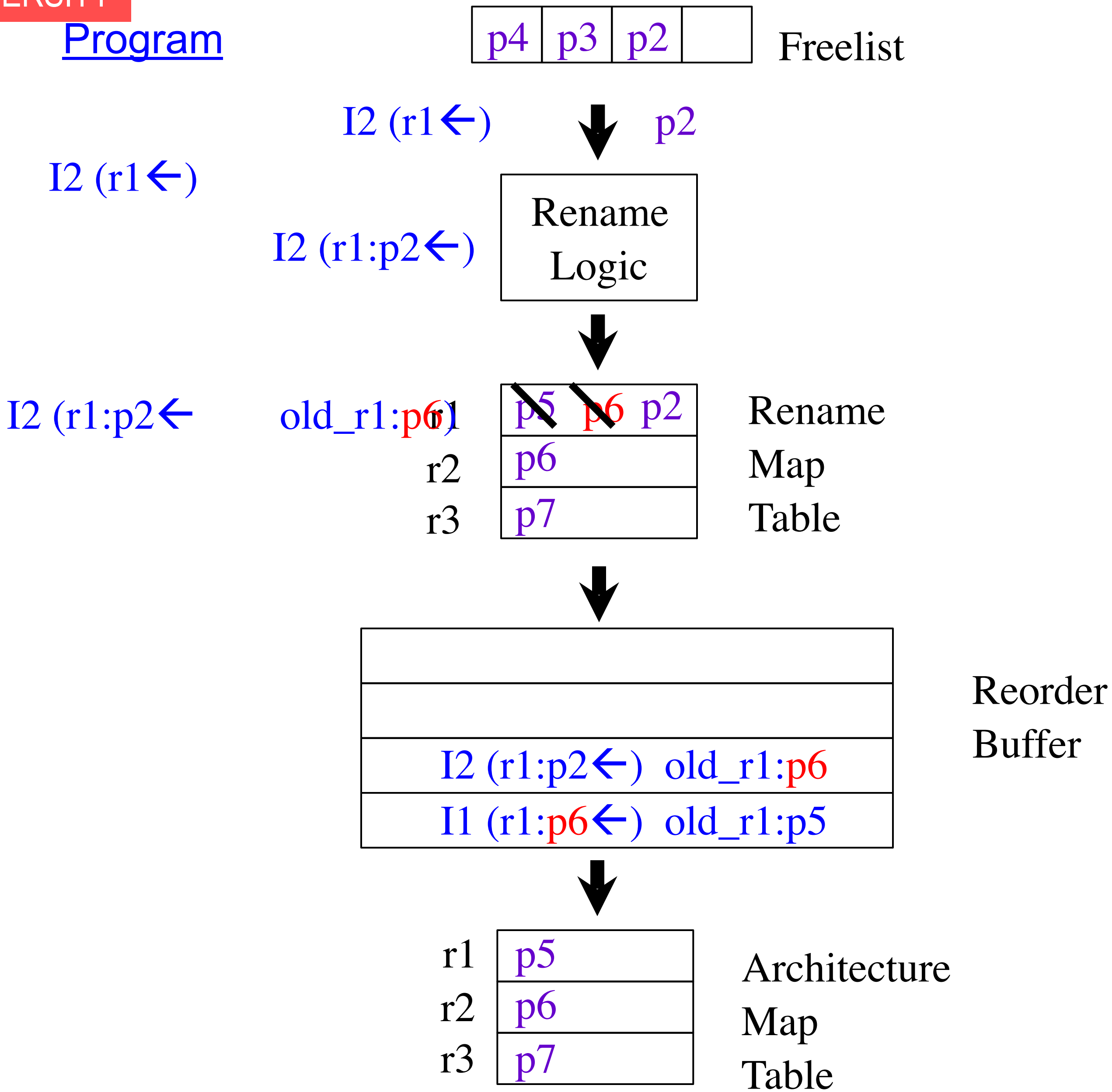
At I3 retirement

Program

I1 (r1←)

I2 (r1←)





Program

I2 (r1←)

I2 (r1:p2←

p4	p3	p2	

Freelist

I2 (r1←)

p2

I2 (r1:p2←)

Rename Logic

old_r1:p6

r1	~~p5~~ ~~p6~~ p2
r2	p6
r3	p7

Rename Map Table

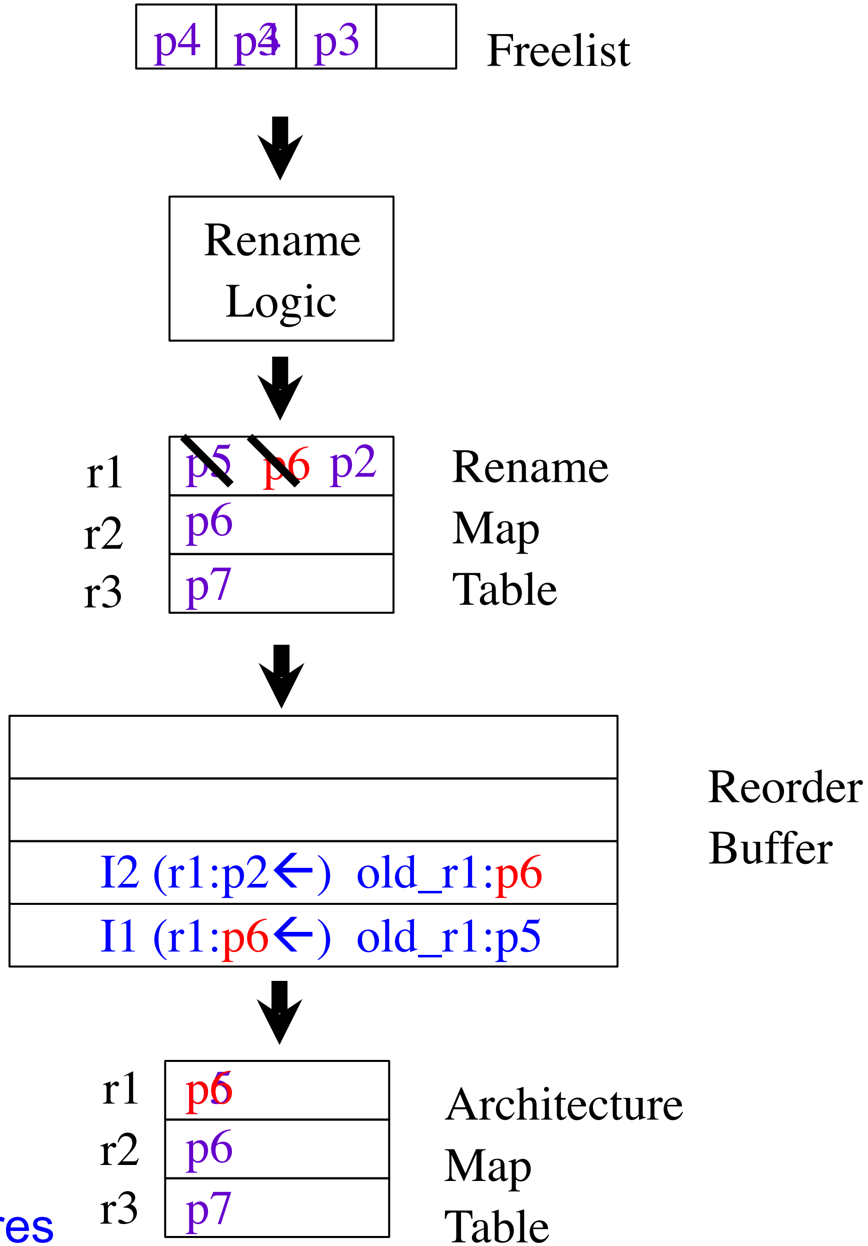
I2 (r1:p2←) old_r1:p6
I1 (r1:p6←) old_r1:p5

Reorder Buffer

r1	p5
r2	p6
r3	p7

Architecture Map Table

Program



p4 | p4 | p3 |

Freelist

Rename Logic

r1 ~~p5~~ ~~p6~~ p2
r2 p6
r3 p7

Rename Map Table

I2 (r1:p2←) old_r1:p6
I1 (r1:p6←) old_r1:p5

Reorder Buffer

r1 p6
r2 p6
r3 p7

Architecture Map Table

RNA prev mapping check

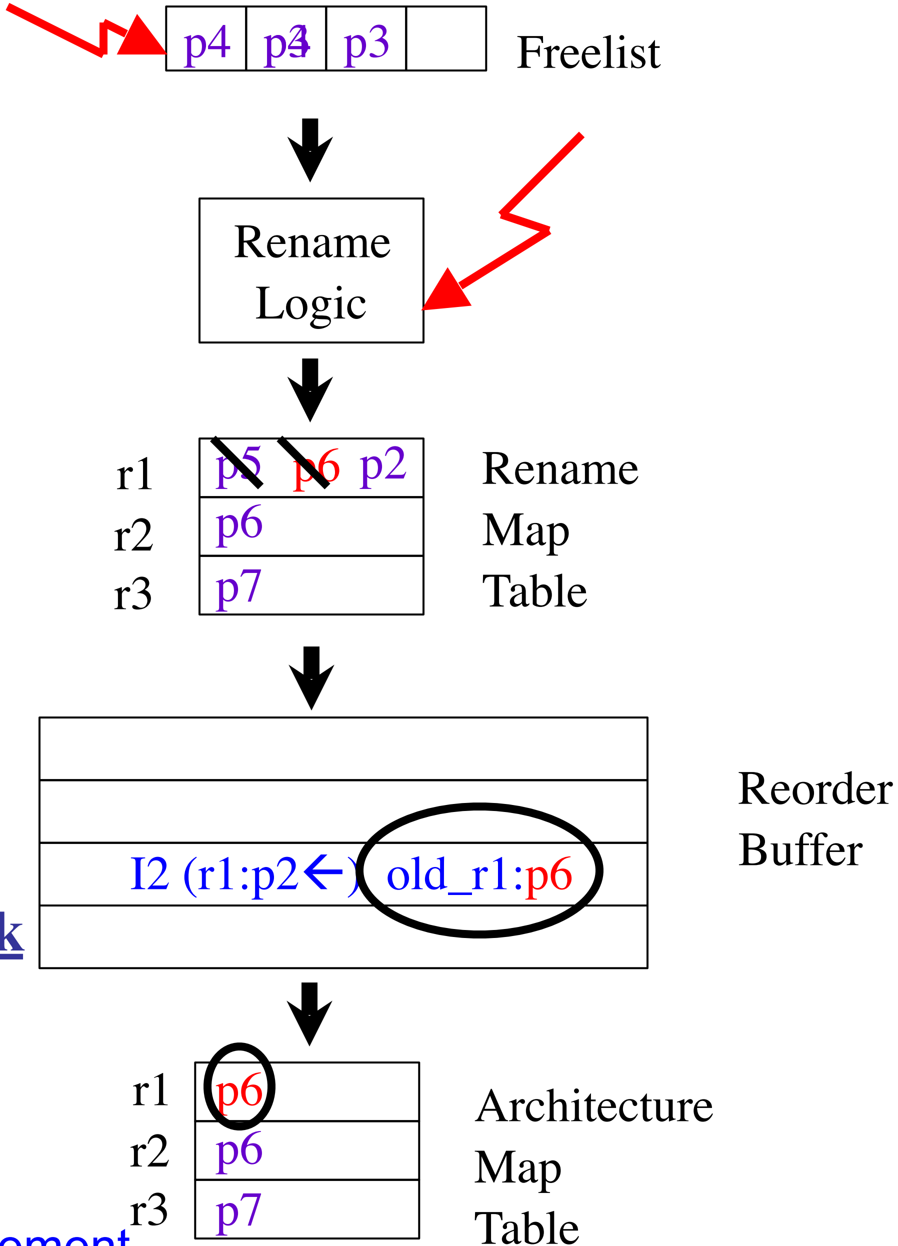
old_r1 == arch_r1?

p5 == p5?

I1 retires

NC STATE UNIVERSITY

Program



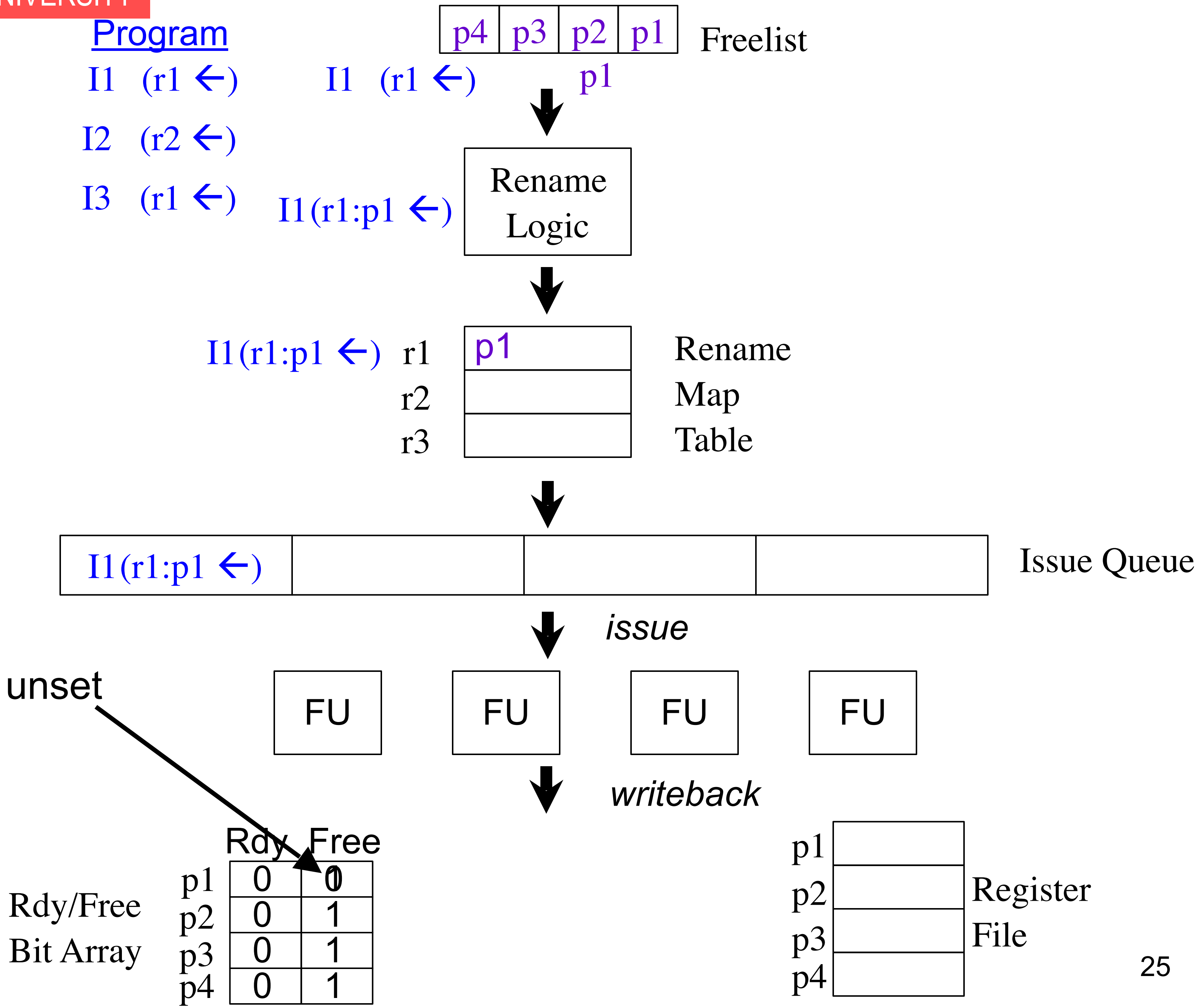
RNA prev mapping check

old_r1 == arch_r1?

p6 == p6?

FAULT NOT DETECTED

At I2 retirement



Program

I1 (r1 ←)

I2 (r2 ←)

I3 (r1 ←)

Freelist: p4 | p3 | p2 | p1

I1 (r1 ←) p1

Rename Logic

I1(r1:p1 ←)

Rename Map Table

I1(r1:p1 ←)

r1	p1
r2	
r3	

Issue Queue

I1(r1:p1 ←)			

issue

FU FU FU FU

writeback

Free bit unset

Rdy/Free Bit Array

	Rdy	Free
p1	0	0
p2	0	1
p3	0	1
p4	0	1

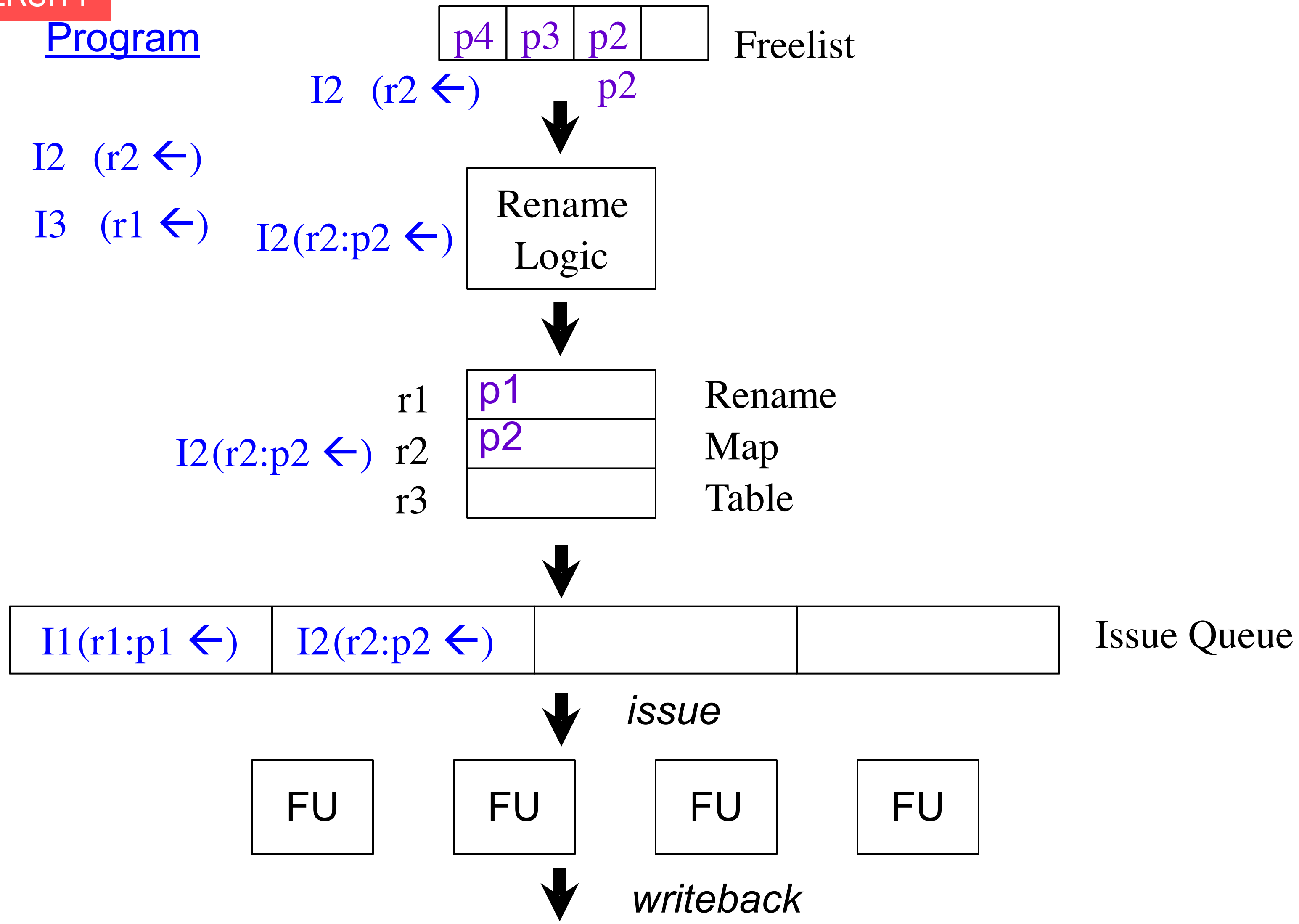
Register File

p1	
p2	
p3	
p4	

Program

I2 (r2 ←)

I3 (r1 ←)



Rdy/Free Bit Array

	Rdy	Free
p1	0	0
p2	0	0
p3	0	1
p4	0	1

Register File

p1
p2
p3
p4

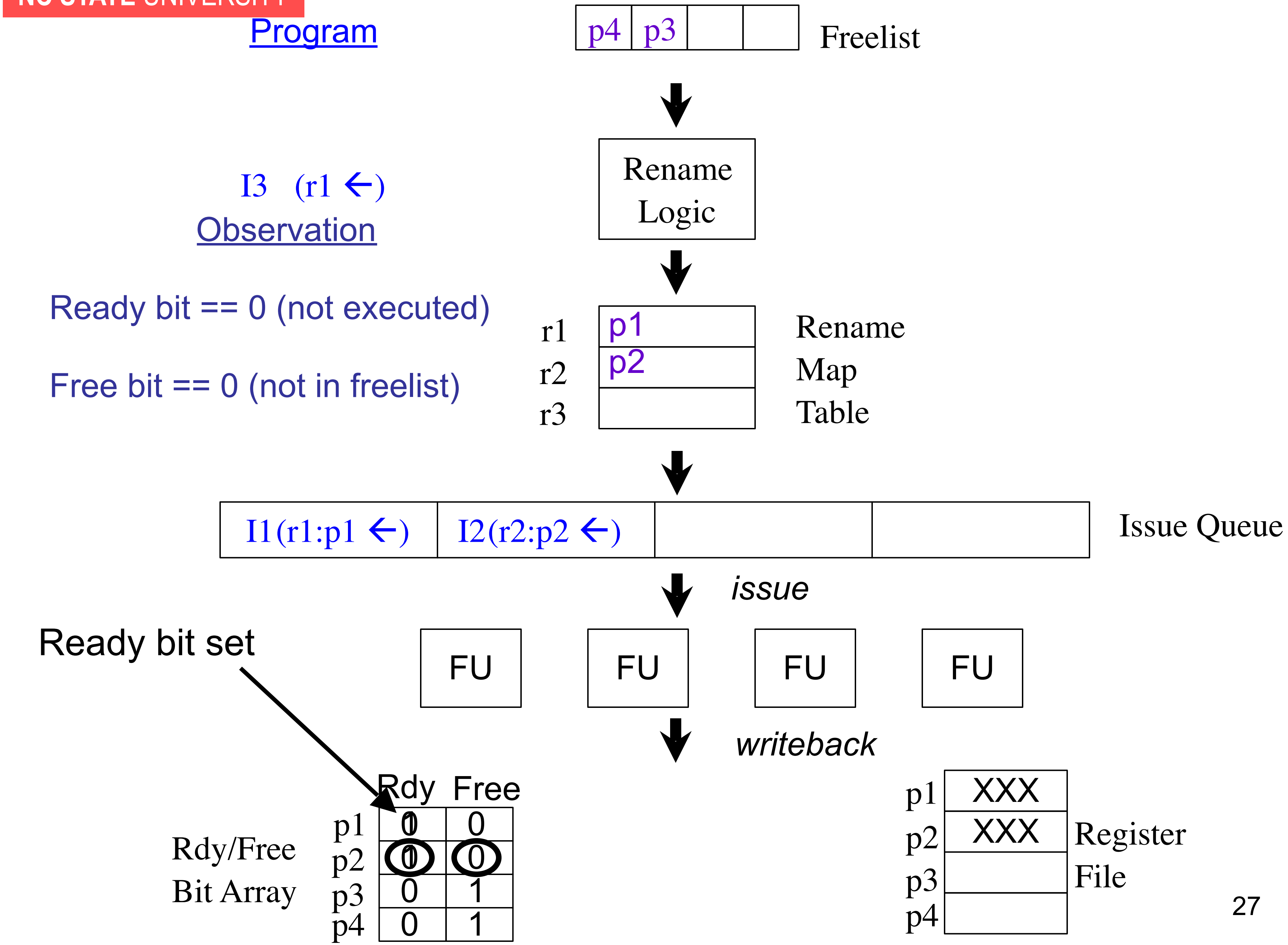
Program

I3 (r1 ←)

Observation

Ready bit == 0 (not executed)

Free bit == 0 (not in freelist)



	Rdy	Free
p1	0	0
p2	0	0
p3	0	1
p4	0	1

p1	XXX
p2	XXX
p3	
p4	

Program

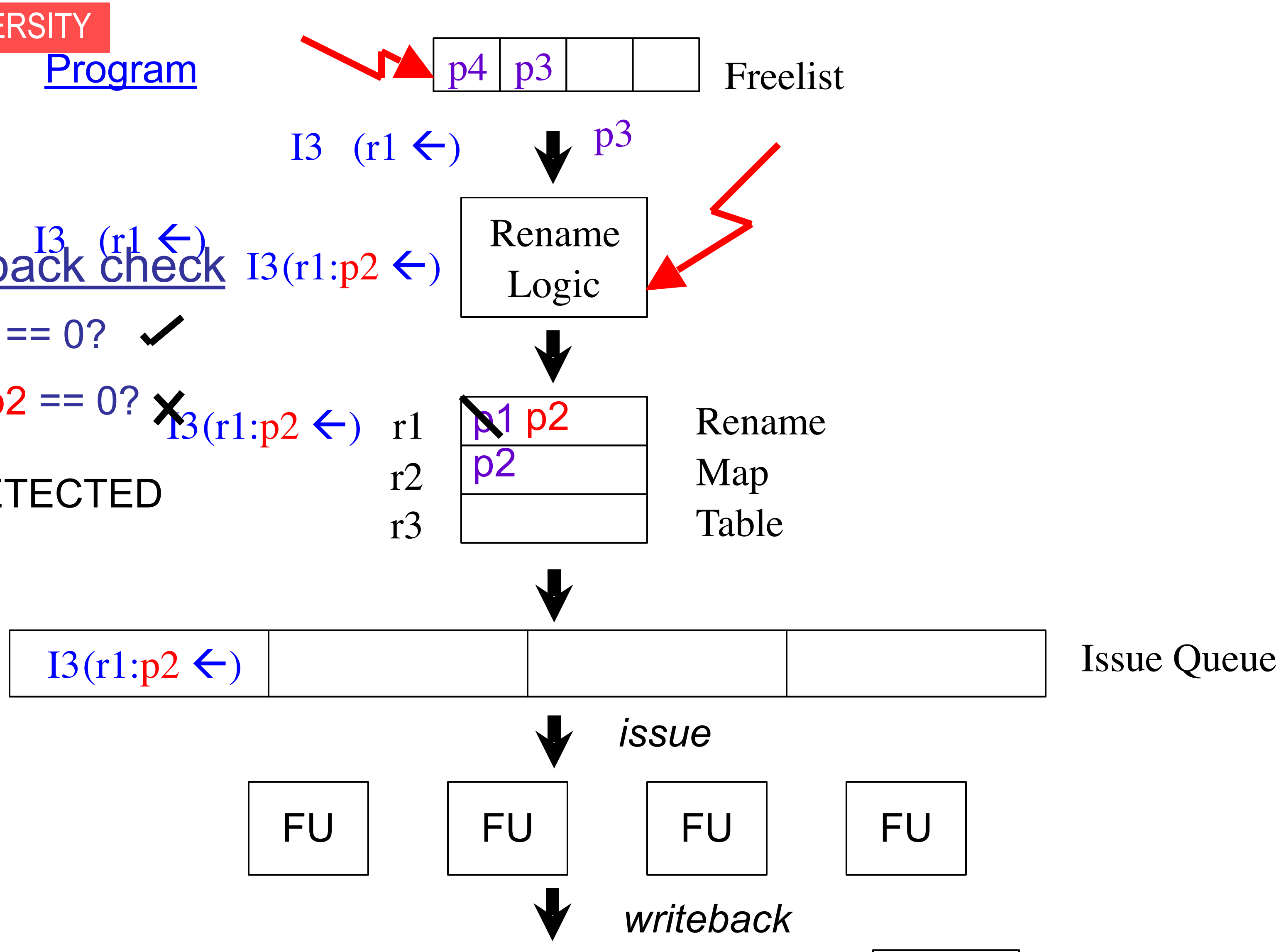
RNA writeback check

I3 (r1 ←)

Free of p2 == 0? ✓

Ready of p2 == 0? ✗

FAULT DETECTED



Rdy/Free Bit Array

	Rdy	Free
p1	1	0
p2	1	0
p3	0	1
p4	0	1

Register File

p1	XXX
p2	XXX
p3	
p4	

RNA summary

- RNA previous mapping check
  - Detects faults in renaming structures
    - Rename map table, Architecture map table
    - Branch checkpoint tables
    - Active list (renaming state)
- RNA writeback state check
  - Detects faults in renaming logic and freelist

Source renaming

- Pure source renaming faults undetected
  - E.g., fault in source renaming logic
  - Researching solutions similar to RNA
- However, faults causing deadlock are detectable
  - Faulty source name causing cyclic dependency
  - Faulty source name points to unpopped freelist entry
  - Other faults that cause phantom producers
- Use watchdog timer to detect deadlocks

Timestamp-based Assertion Checking (TAC)

- Confirm data dependent instructions issued sequentially
  - Assign timestamps to instructions at issue
  - At retirement, confirm instruction timestamp greater than producers' timestamps
- TAC check

  Instruction timestamp >= Producer's timestamp + latency
- Faults on checking logic can only cause false alarms

Dataflow

I1 (r1←)

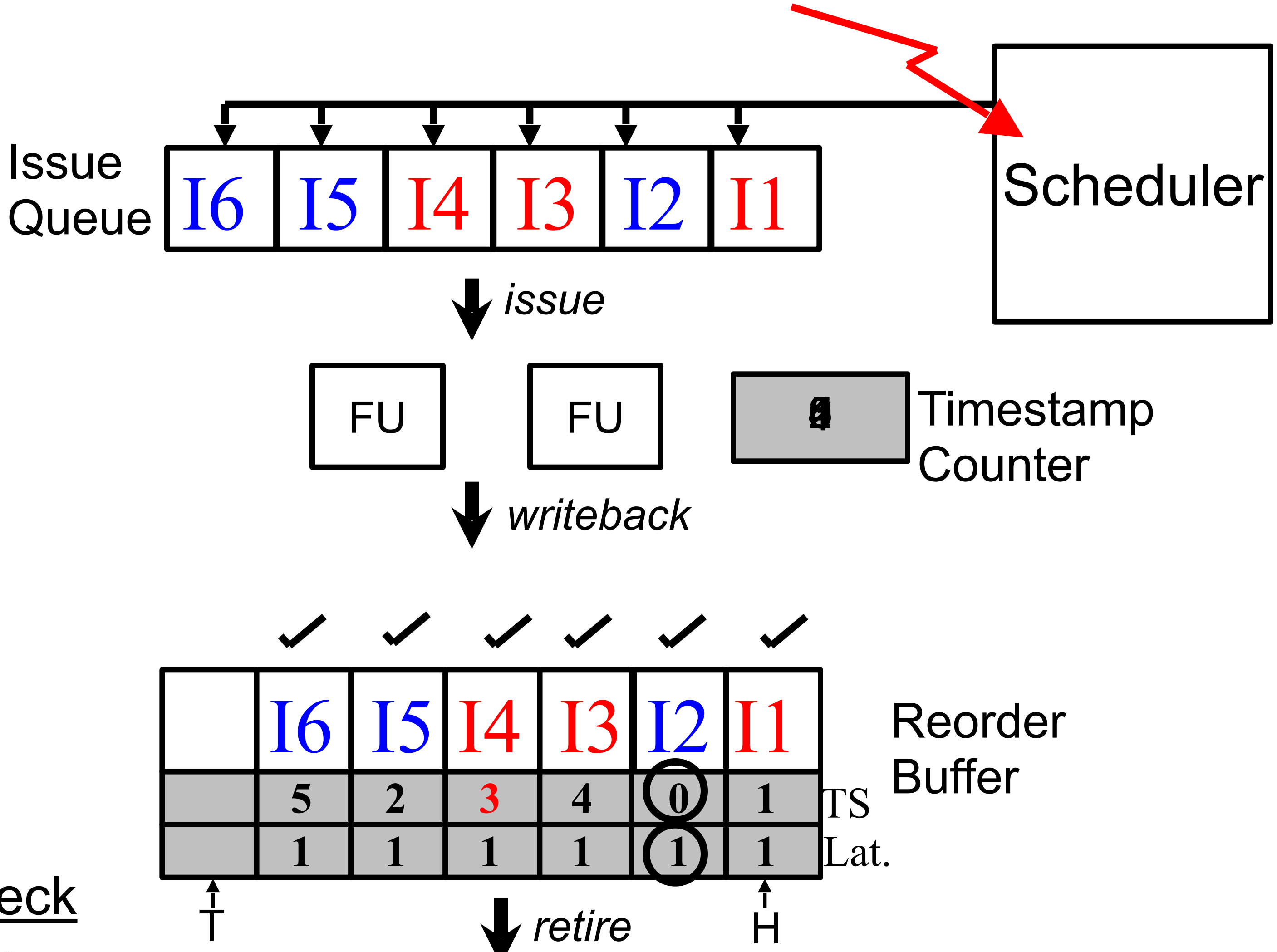
I3 (r1←, ←r1)

I4 (r1←, ←r1)

I2 (r3←)

I5 (r3←, ←r3)

I6 (r3←, ←r3)



	I6	I5	I4	I3	I2	I1	
	✓	✓	✓	✓	✓	✓	
	5	2	3	4	0	1	TS
	1	1	1	1	1	1	Lat.

TAC check

Instr TS >= Prod TS + Prod Lat

	TS	Lat.
r1	0	0
r2	0	0
r3	0	0

Architectural State

Dataflow

I1 (r1←)

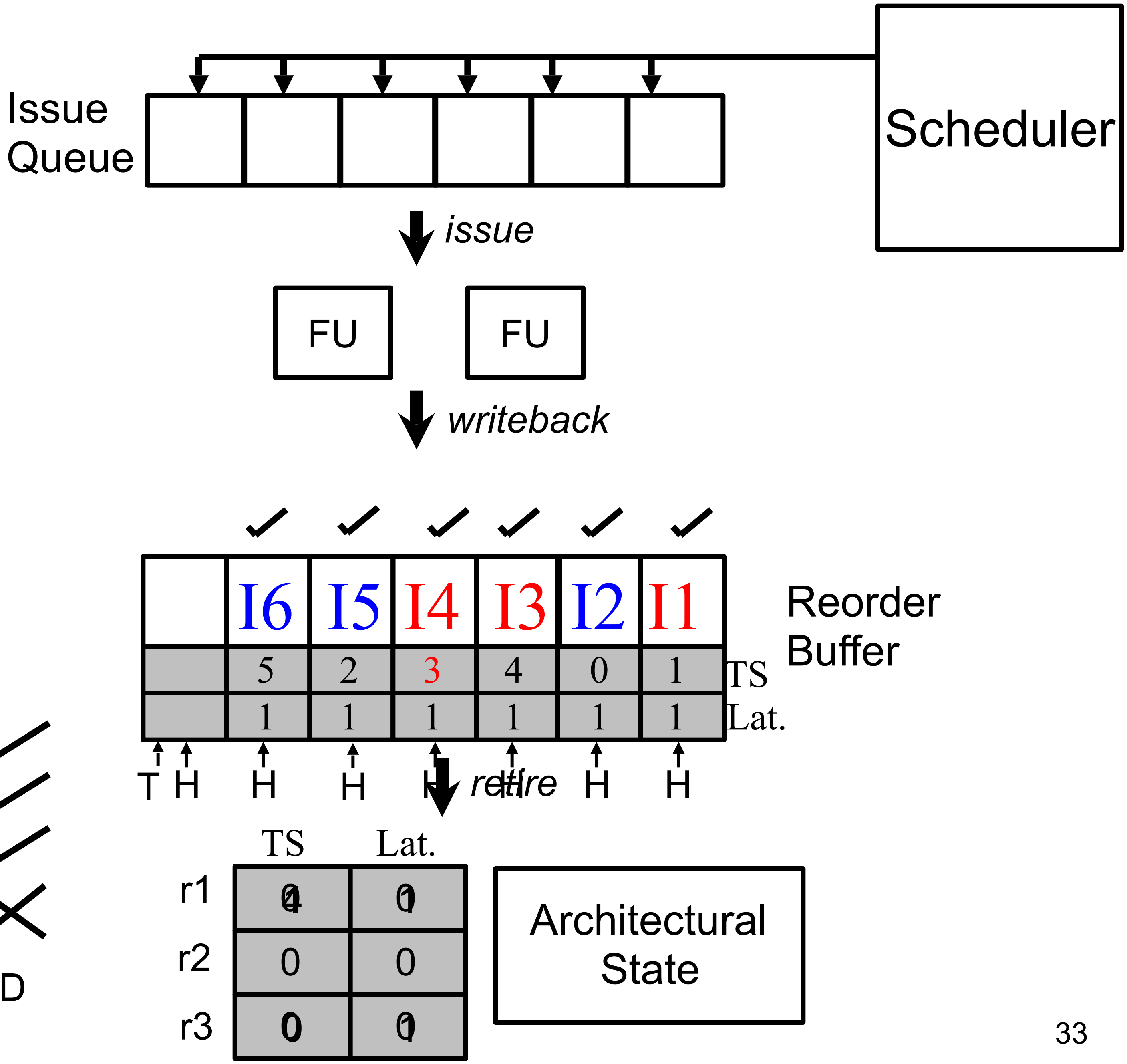
I3 (r1←, ←r1)

I4 (r1←, ←r1)

I2 (r3←)

I5 (r3←, ←r3)

I6 (r3←, ←r3)



	I6	I5	I4	I3	I2	I1
	5	2	3	4	0	1	TS
	1	1	1	1	1	1	Lat.

	TS	Lat.
r1	4	1
r2	0	0
r3	0	1

TAC check

I1: 1 >= 0 + 0 ✓

I2: 0 >= 0 + 0 ✓

I3: 4 >= 1 + 1 ✓

I4: 3 >= 4 + 1 ✗

FAULT DETECTED

Experiments

- Randomly injected faults in timing simulator
  - 1000 faults per benchmark
  - Faults target issue and rename state
  - Simulation ends 1 million cycles following fault injection
- Observations
  - Fault detected by an assert (Assert) or not (Undet)
  - Fault corrupts architectural state (SDC) or not (Masked)
- Possible outcomes
  - Assert+SDC
  - Undet+SDC
  - Assert+Masked
  - Undet+Masked

TAC - Fault Injection Experiments

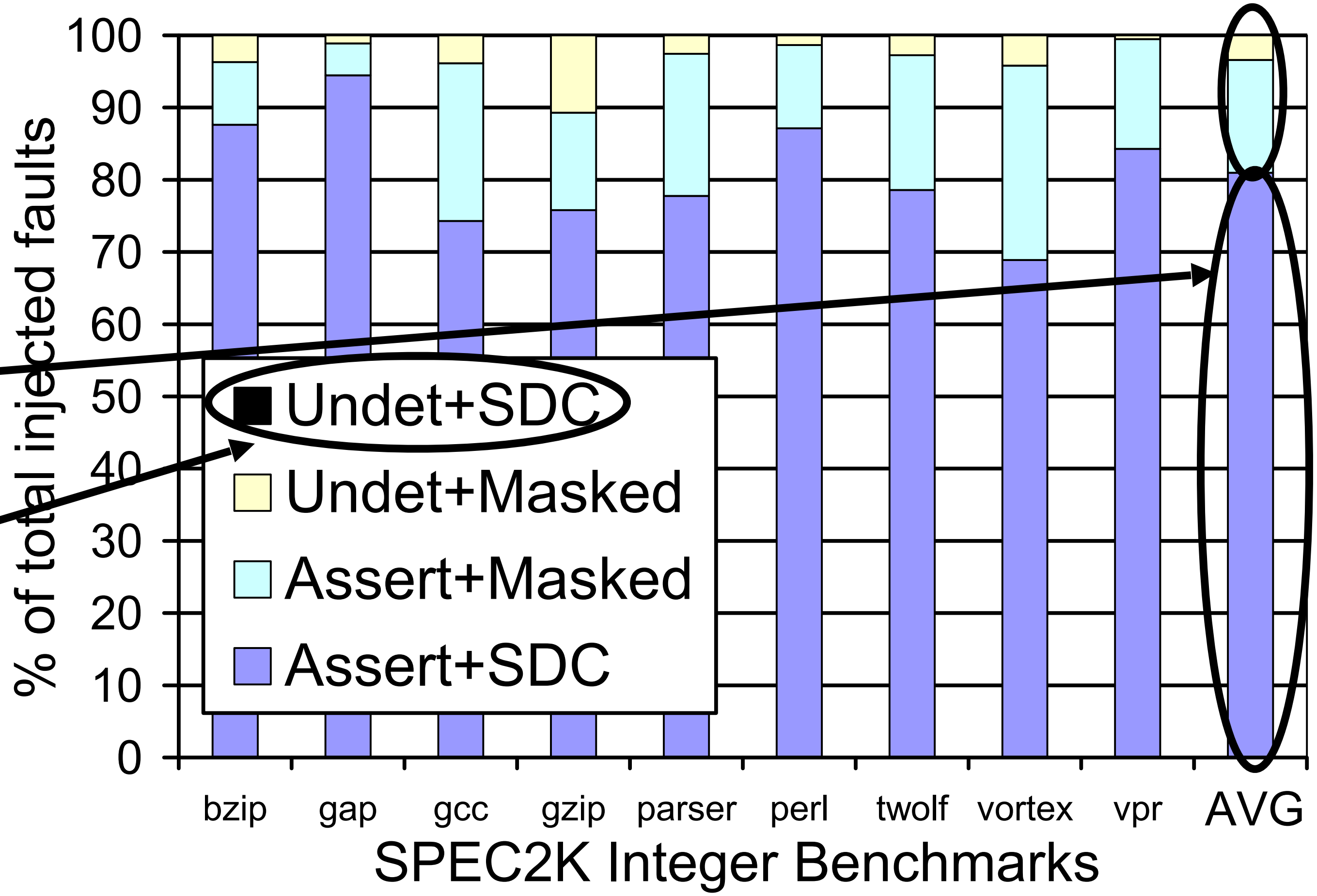
- Type of faults injected
  - Ready bits prematurely set
  - Speculatively issued cache-missing load dependents not reissued

TAC - Fault Injection Experiments

80% of faults cause SDC
20% of faults are masked

All SDC faults are detected

Zero undetected SDC faults



RNA - Fault Injection Experiments

- Faults injected
  - Flip bits of an entry in architecture map table
  - Flip bits of an entry in rename map table
  - Flip bits of an entry in freelist
  - Flip destination register bits at dispatch
- Watchdog timer included to detect deadlocks
- Additional possible outcomes
  - Assert+Wdog : RNA detected a future deadlock
  - Undet+Wdog : Deadlock undetected by RNA (possibly would have been caught by RNA in future)

RNA - Fault Injection Experiments

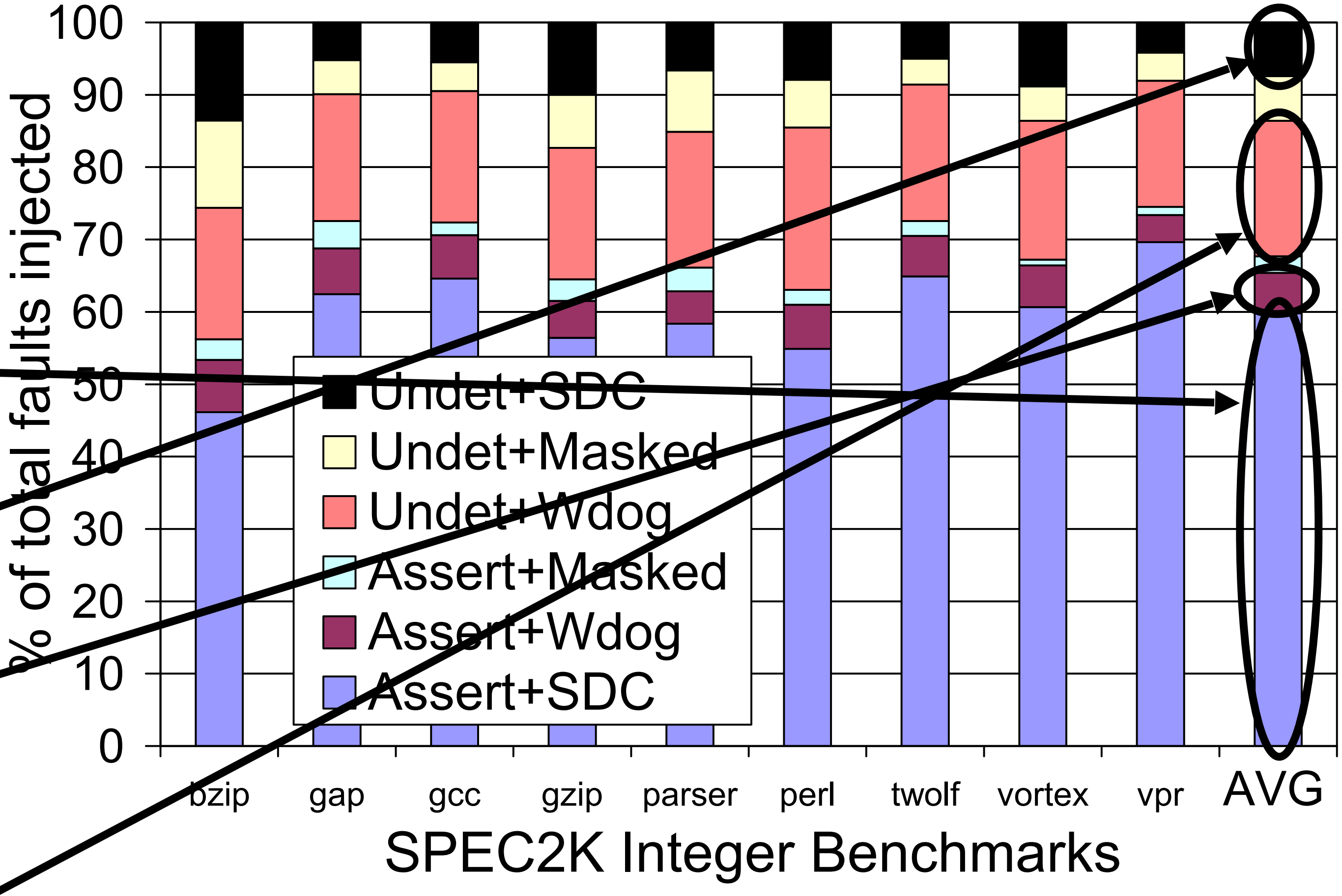
68% SDC faults
24% deadlocks
8% masked faults

88% of SDC faults detected by RNA

12% of SDC faults undetected

25% of deadlocks detected by RNA beforehand

75% of deadlocks not detected by RNA (but detected by watchdog)



RNA - Fault outcome distribution

“dest” faults cause deadlocks because of phantom producers

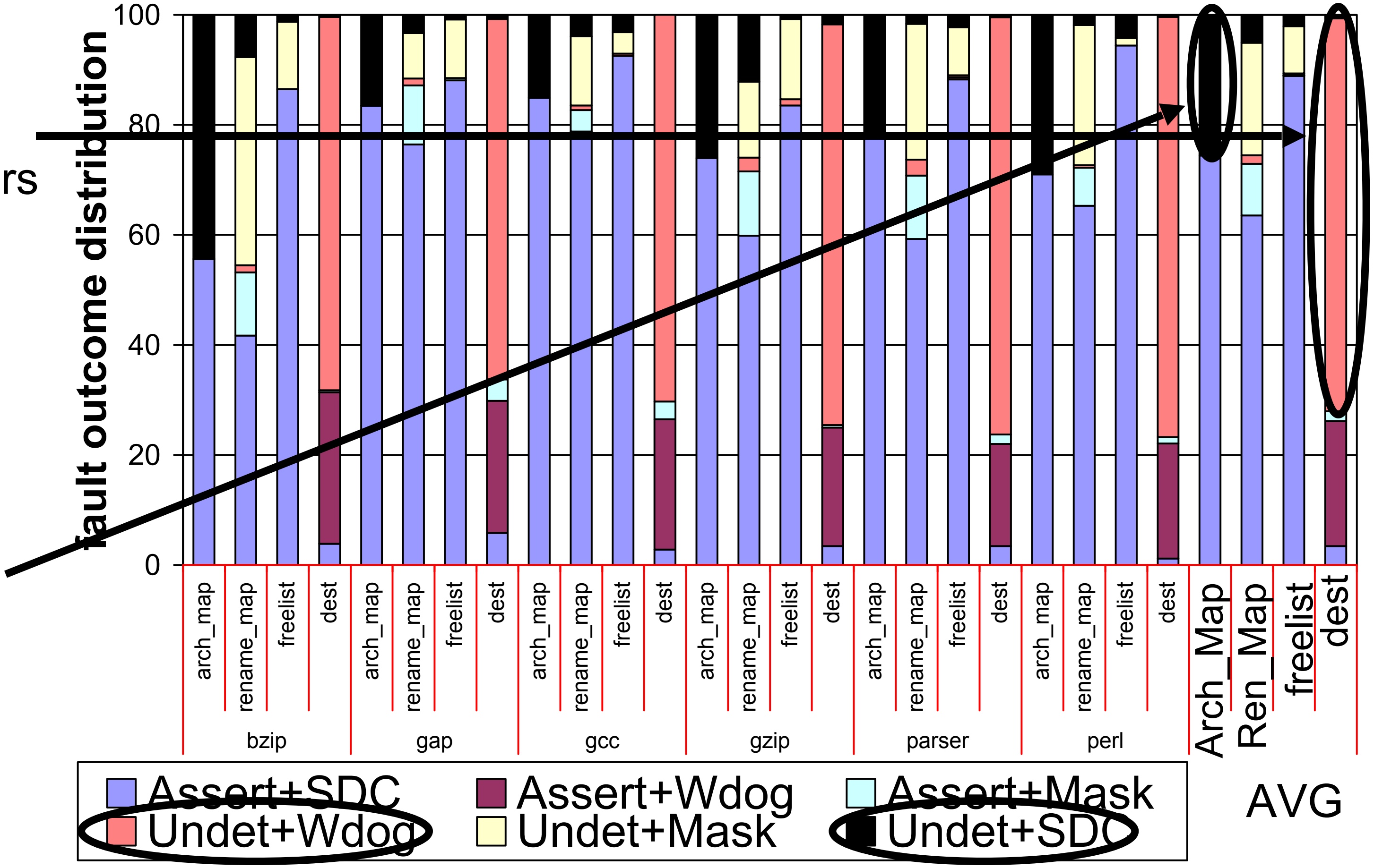
Deadlock blocks retirement

Thus, RNA check can’t complete

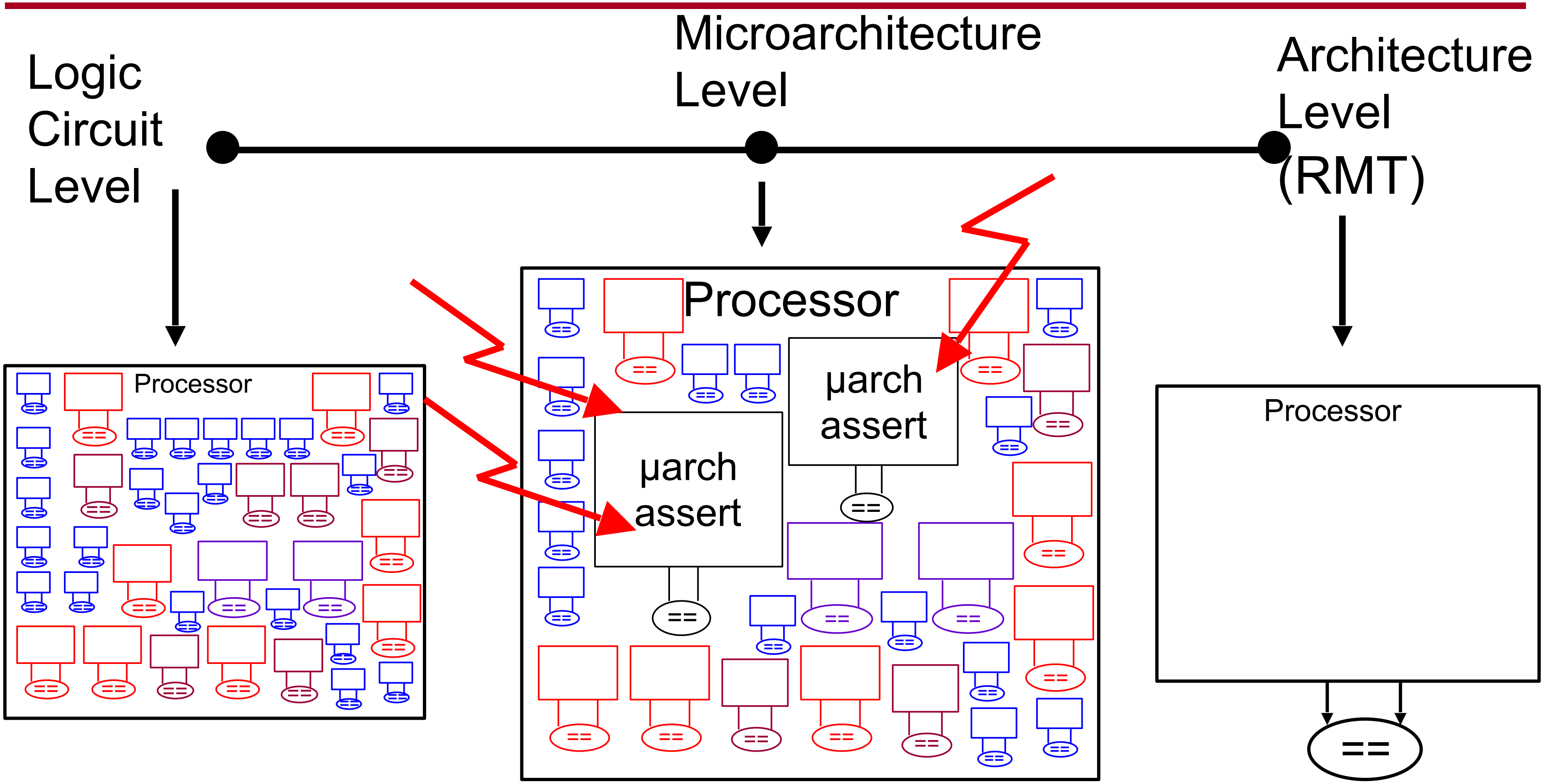
“arch_map” faults cause most undetected SDC

Long live register range

System trap before RNA check can complete



Conclusions



Logic Circuit Level

Microarchitecture Level

Architecture Level (RMT)