

# **6.823 SPring15**

## **Quiz 2 Review (L09-L14)**

**TA: Po-An Tsai, Hsin-Jung Yang**

# Lecture 9-14 are all about ILP

- Instruction-level parallelism (ILP)
  - Execute as many instructions as possible at the same time to maximize throughput and hide long latency by memory/ALU
- Why this is not easy?
  - Dependencies between instructions

# Dependency

- Data dependency
  - RAW WAR WAW
- Control dependency
  - Branch jump
- Structural dependency
  - Only one ALU

# Ways to Solve Dependency

- For false dependency
  - Indirection
- For true dependency
  - Stall
  - Bypass
  - Speculate => best if you speculate mostly correctly

# Ways to Solve Dependency

- Data dependency
  - RAW WAR WAW
  - => Scoreboard (L9) ROB, Register renaming (L10), Store queue(L13)
- Control dependency
  - Branch jump
  - => Branch prediction (L11)
- Structural dependency
  - Only one ALU
  - => Superscalar (L9)

# Speculation

- We speculate a lot!
  - Branch
  - No exceptions
  - Addresses for load/store are not the same
- How to manage old and new values?
  - Greedy/lazy (L12)

# Mis-speculation

- Recovery according to your policy(L12)
  - Snapshot
  - Rollback

# Hide Long Latency

- A cache miss takes 100 cycles
- A divide takes 20 cycles
  - Execute following instructions, but hold them until those long-latency instruction finish.
- Use Little's law to calculate how many instruction in flight



# Hide Long Latency

- A cache miss takes 100 cycles
- A divide takes 20 cycles
  - Switch to another independent thread until they finish (L14)
- Use Little's law to calculate how many threads needed

# Example OoO Pipeline

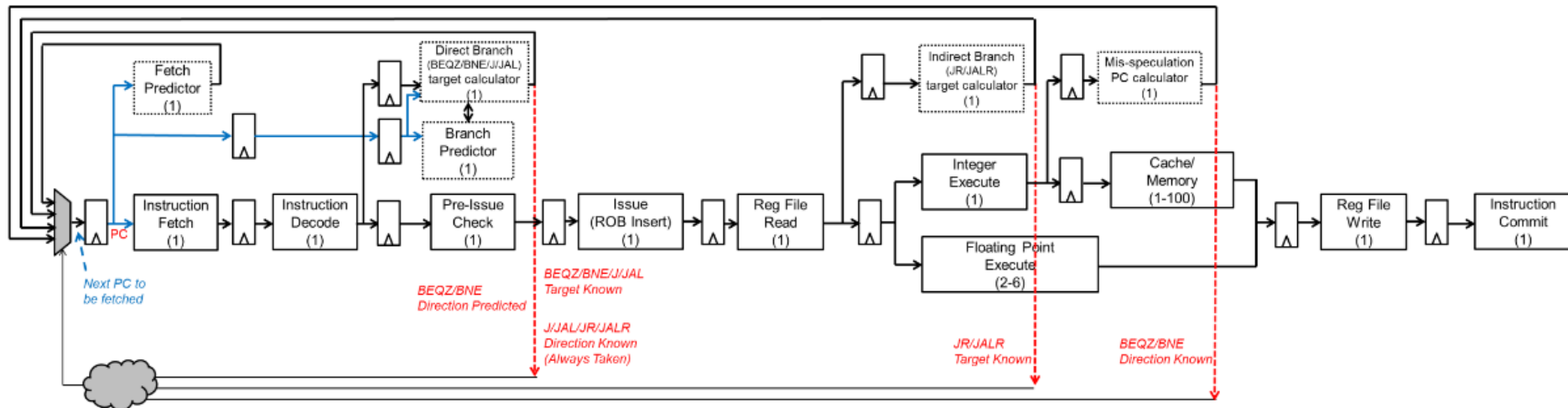
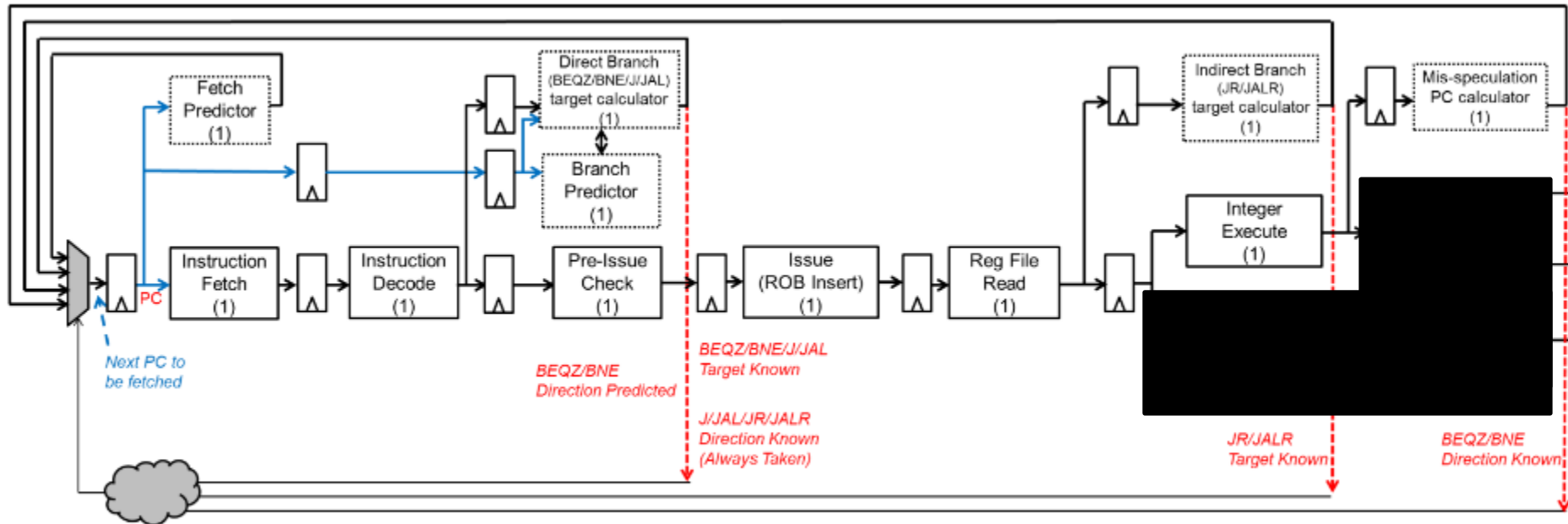


Figure 1: Out-of-order Pipeline

# Branch Prediction



**Figure 1: Out-of-order Pipeline**

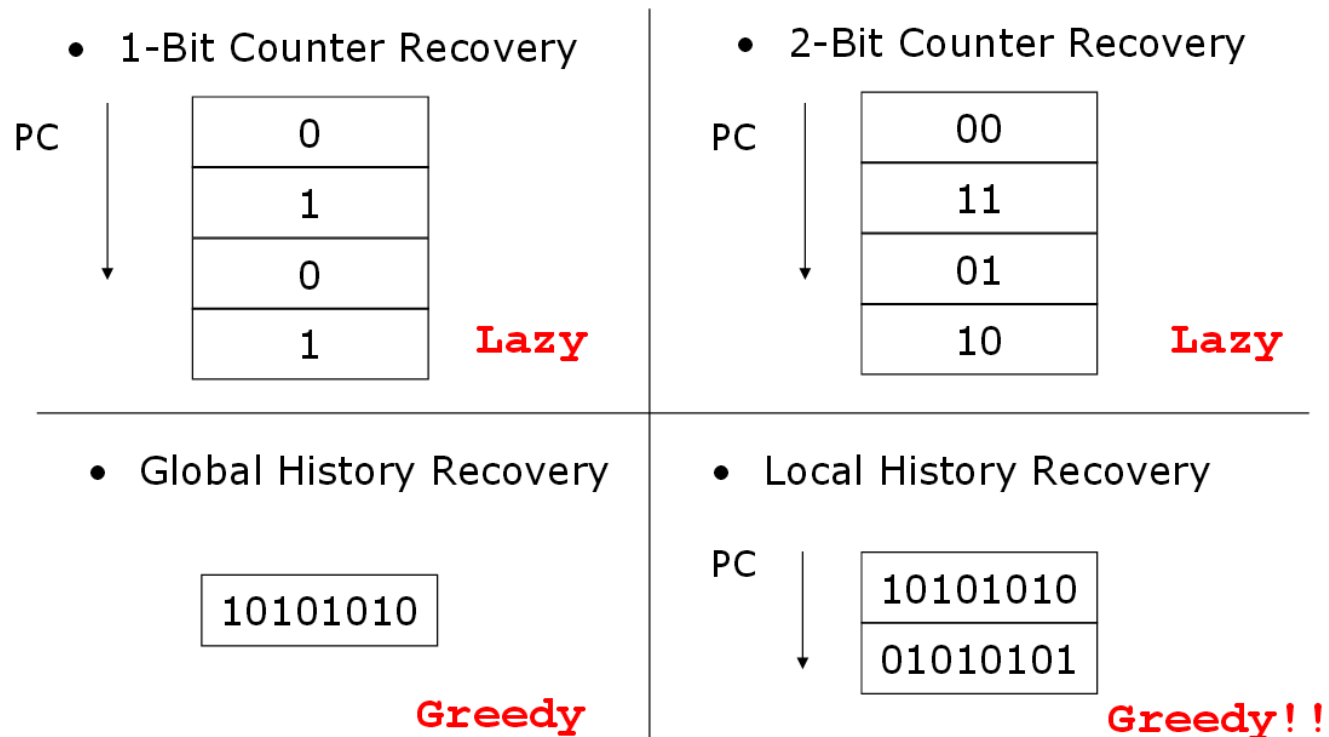
# Branch Prediction

- Speculate what the next instruction is
  - Static
  - 1 bit predictor
  - 2 bit predictor
  - Global history (a history register and lots of predictor)
  - Local history (many history registers and lots of predictor)
  - Combined local and global history

# Branch Prediction

- Data management

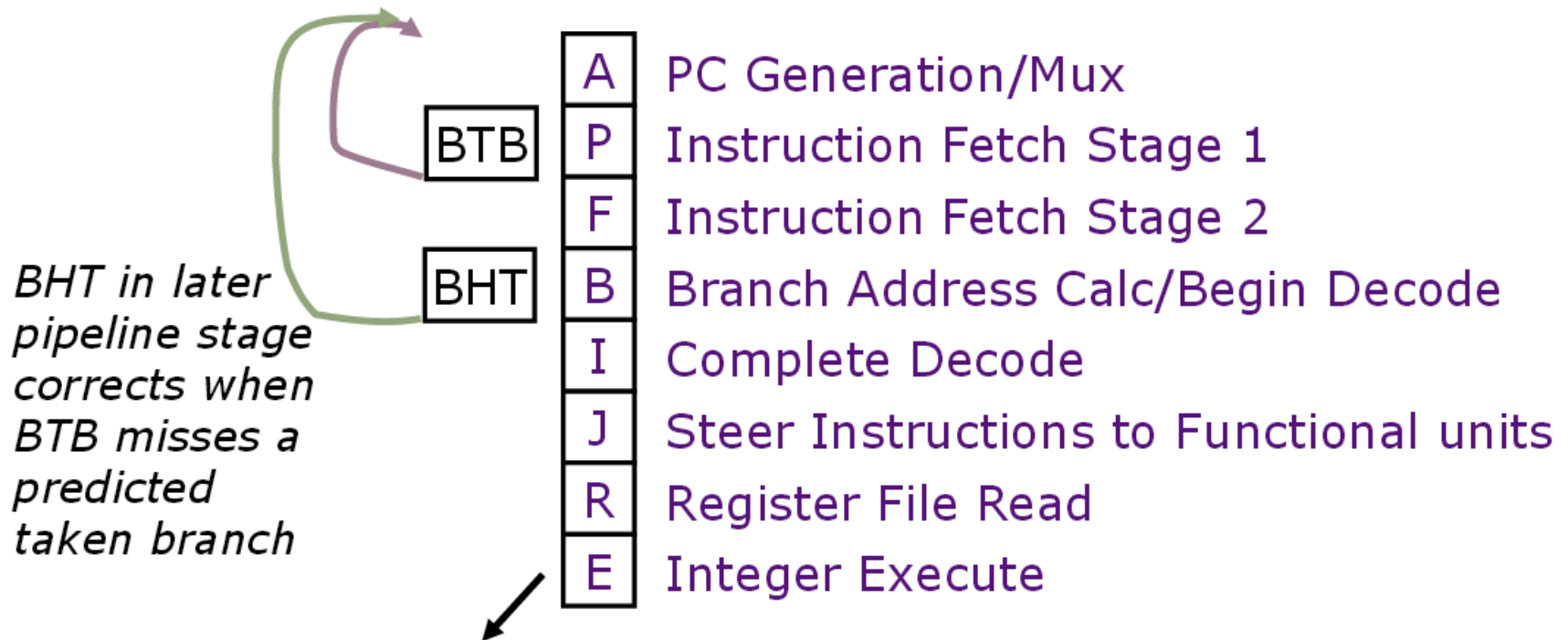
## Branch Predictor Recovery



# Branch Target Buffer

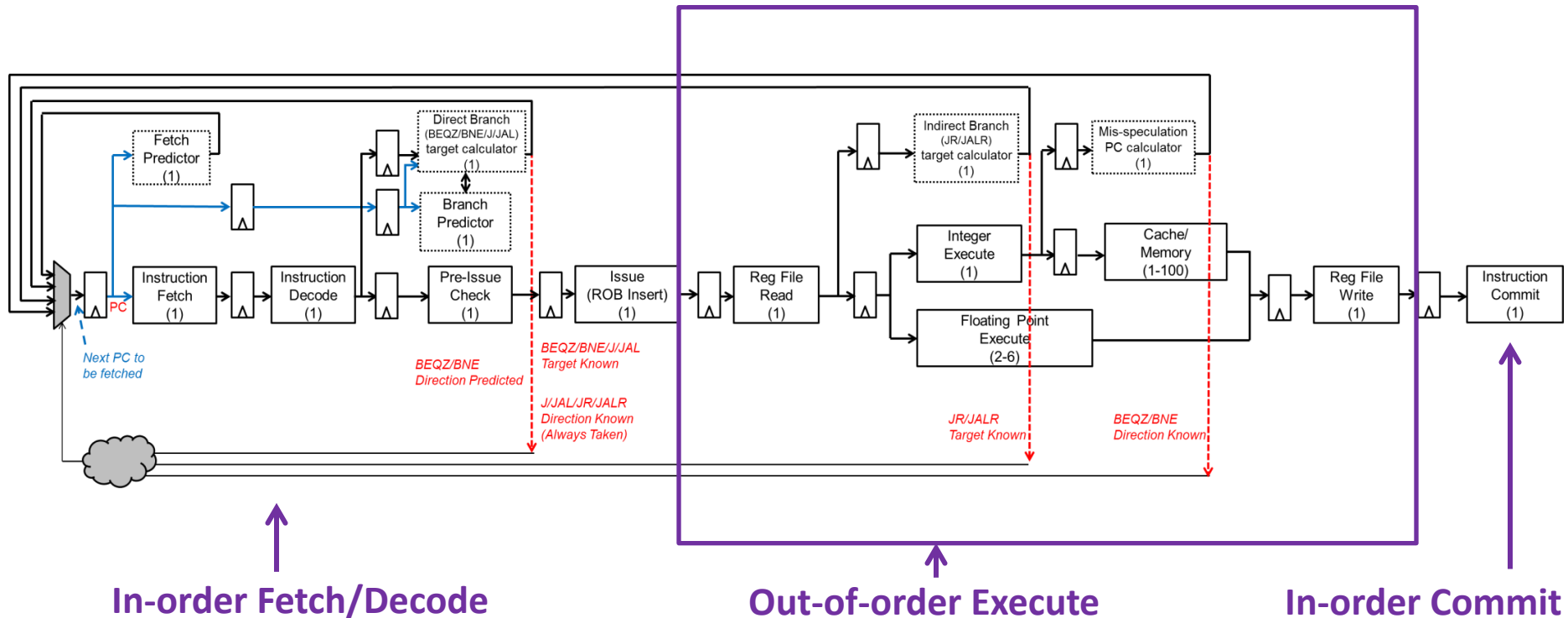
- Store the next PC of branch/jmp instructions seen last time
- Get address earlier than predictor

# Combine BTB and Predictor(BHT)



*BTB/BHT only updated after branch resolves in E stage*

# Out-of-Order Execution



When can we execute an instruction out-of-order?

**Need to consider data dependency**  
(register dependency, memory dependency)




# Data Dependency

- Register Dependency

Data-dependence


$r_3 \leftarrow (r_1) \text{ op } (r_2)$   
 $r_5 \leftarrow (r_3) \text{ op } (r_4)$



Read-after-Write  
(RAW) hazard

Anti-dependence


$r_3 \leftarrow (r_1) \text{ op } (r_2)$   
 $r_1 \leftarrow (r_4) \text{ op } (r_5)$



Write-after-Read  
(WAR) hazard

Output-dependence

$r_3 \leftarrow (r_1) \text{ op } (r_2)$   
 $r_3 \leftarrow (r_6) \text{ op } (r_7)$




Write-after-Write  
(WAW) hazard

# Data Dependency

- Register Dependency

Data-dependence


$r_3 \leftarrow (r_1) \text{ op } (r_2)$   
 $r_5 \leftarrow (r_3) \text{ op } (r_4)$



Read-after-Write  
(RAW) hazard

Anti-dependence


$r_3 \leftarrow (r_1) \text{ op } (r_2)$   
 $r_1 \leftarrow (r_4) \text{ op } (r_5)$



Write-after-Read  
(WAR) hazard

Output-dependence

$r_3 \leftarrow (r_1) \text{ op } (r_2)$   
 $r_3 \leftarrow (r_6) \text{ op } (r_7)$



Write-after-Write  
(WAW) hazard

Handled by register renaming

Handled by register renaming

# Data Dependency

- Memory Dependency

st r1, 4(r2)

ld r3, 8(r4)

When is the load dependent on the store?

**When  $(r2 + 4) == (r4 + 8)$**

Do we know this issue when the instruction is decoded? **No**

# Data Dependency

- Memory Dependency

st r1, 4(r2)

ld r3, 8(r4)

## Solution:

(1) **Stall:** can execute load before store only if the addresses are known to be different

(2) **Address speculation:** guess  $r2+4 \neq r4+8$  and  
execute load before store

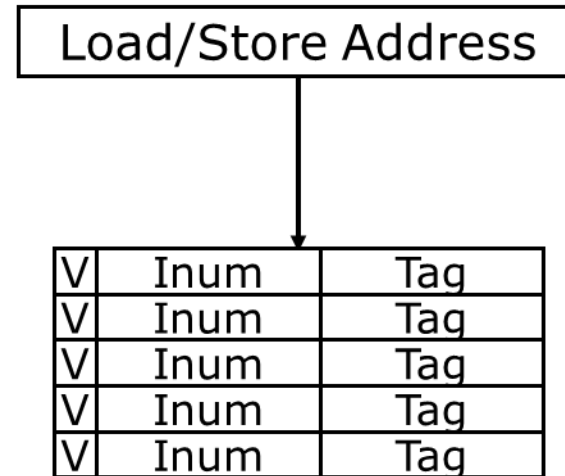
 **Speculative Load Buffer**

# Data Dependency

- Speculative Load Buffer

**Speculation check:**

Detect if a load has executed before an earlier store to the same address – missed RAW hazard



**On load execute:** mark entry valid, instruction number and tag

**On load commit:** clear valid bit

**On load abort:** clear valid bit

**On store execute:** if tag matches and the instruction is younger than the store -> Abort!

# Speculative Data Management

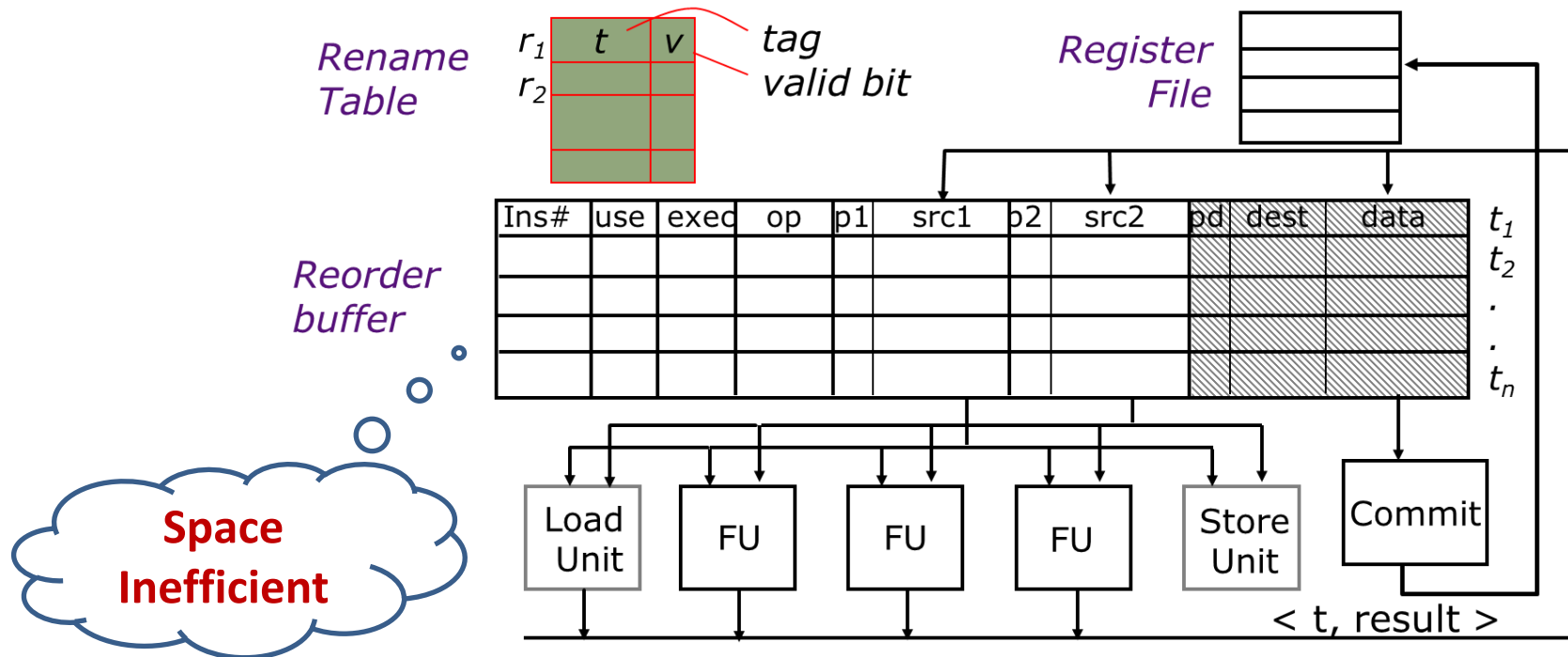
- **When do we do speculation?**
  - Branch prediction
  - Assume no exceptions/interrupts
  - Assume no memory dependency
  - ...
- **How do we manage speculative data?**
  - **Greedy (or Eager) Update**
    - Update the value in place
    - Maintain a log of old values to use for recovery
  - **Lazy Update**
    - Buffer the new value and leave the old value in place
    - Replace the old value only at 'commit' time

# Speculative Data Management

- **Type of speculative data**
  - **Branch prediction**
    - history registers, prediction counters (see P.13)
  - **Register values**
    - Lazy update: store new values in the ROB and update registers during commit
    - Hybrid: store both new and old values in the unified physical register file
  - **Store values to memory**
    - Laze update: store new values in the speculative store buffer and write to non-speculative store buffer/cache/memory during commit

# Speculative Data Management

- **Register Value Management**
  - Approach 1: store new values in the ROB and update registers during commit





# Speculative Data Management

- **Register Value Management**
  - Approach 2: keep all data values in a physical register file

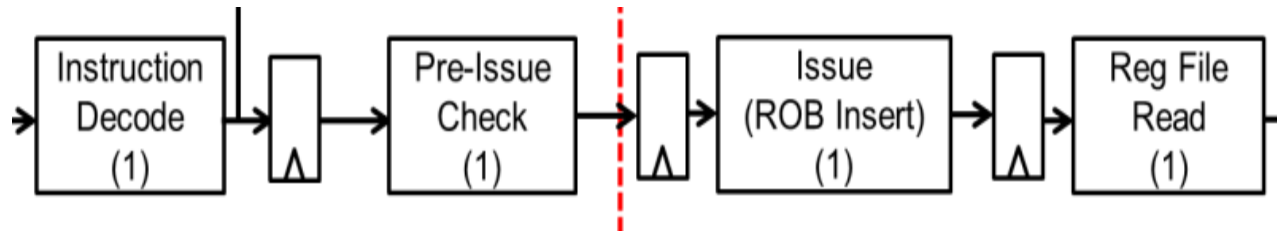
Rename Table		Physical Regs		Free List
R0		P0		P0
R1	P8	P1		P1
R2		P2		P3
R3	P7	P3		P2
R4		P4		P4
R5		P5	<R6> p	
R6	P5	P6	<R7> p	
R7	P6	P7	<R3> p	
		P8	<R1> p	
		Pn		

ROB

use	ex	op	p1	PR1	p2	PR2	Rd	LPRd	PRd



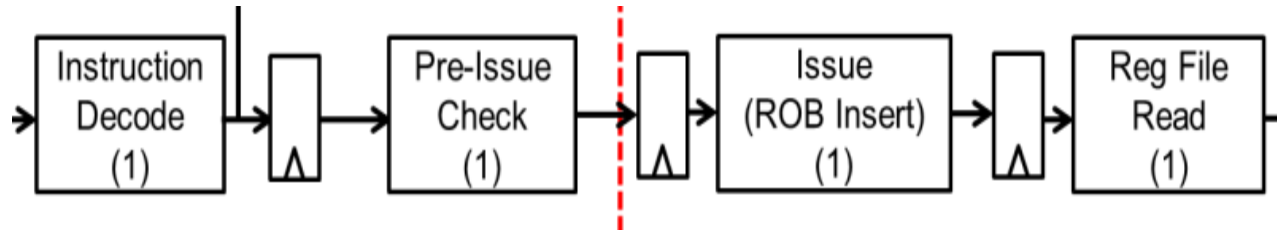
# Out-of-Order Execution



## Pre-Issue Check:

- The ROB is checked for available slots
- The free list is checked for free rename registers (if necessary)
- For store instructions, the non-speculative store buffer is checked for available slots

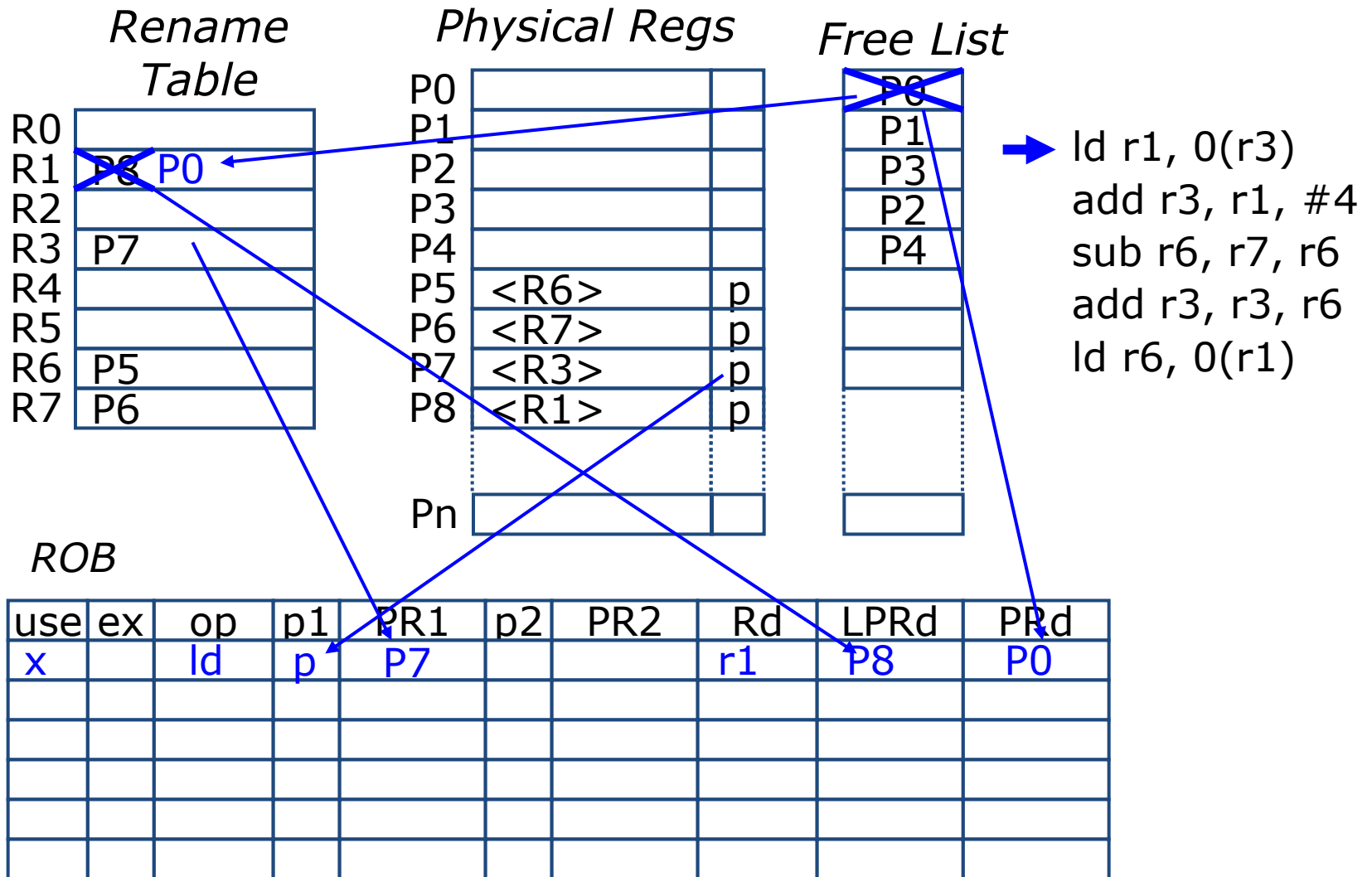
# Out-of-Order Execution



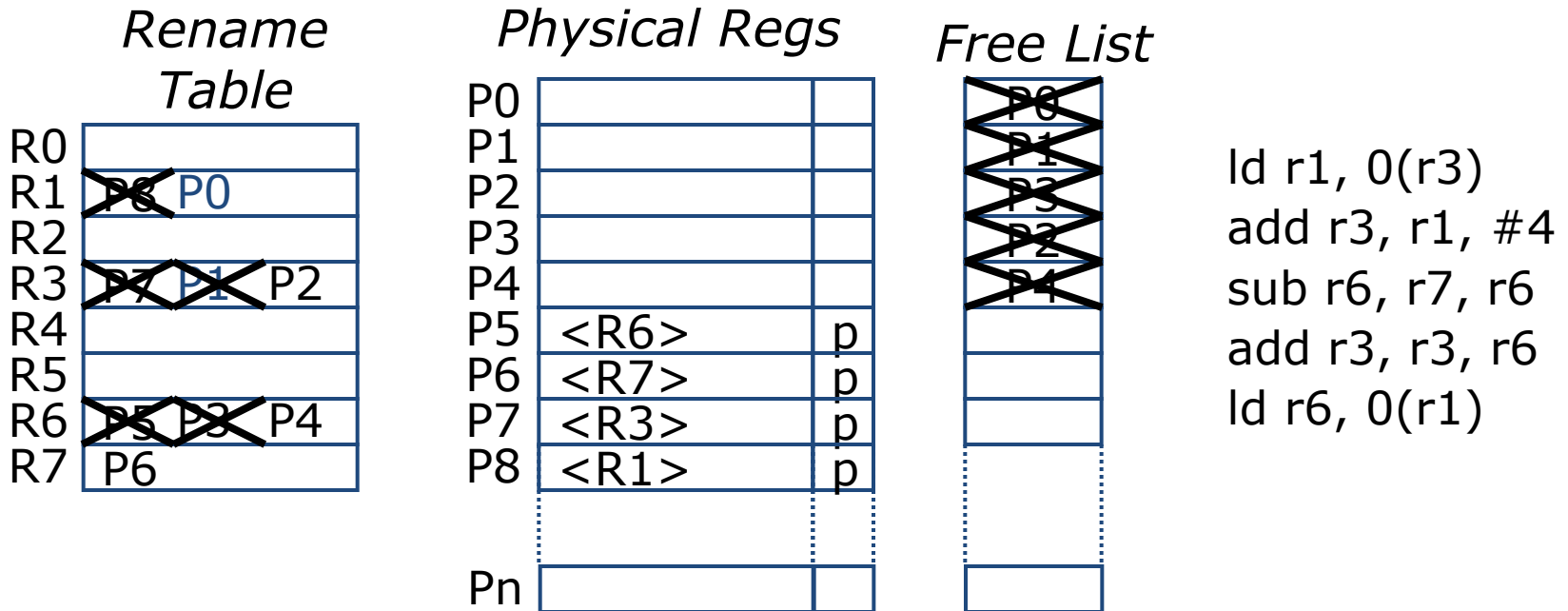
## ROB Insert:

- The instruction is inserted into the ROB only if all the checks in the previous cycle (Pre-Issue check) pass
- The destination register is renamed

# Out-of-Order Execution



# Out-of-Order Execution

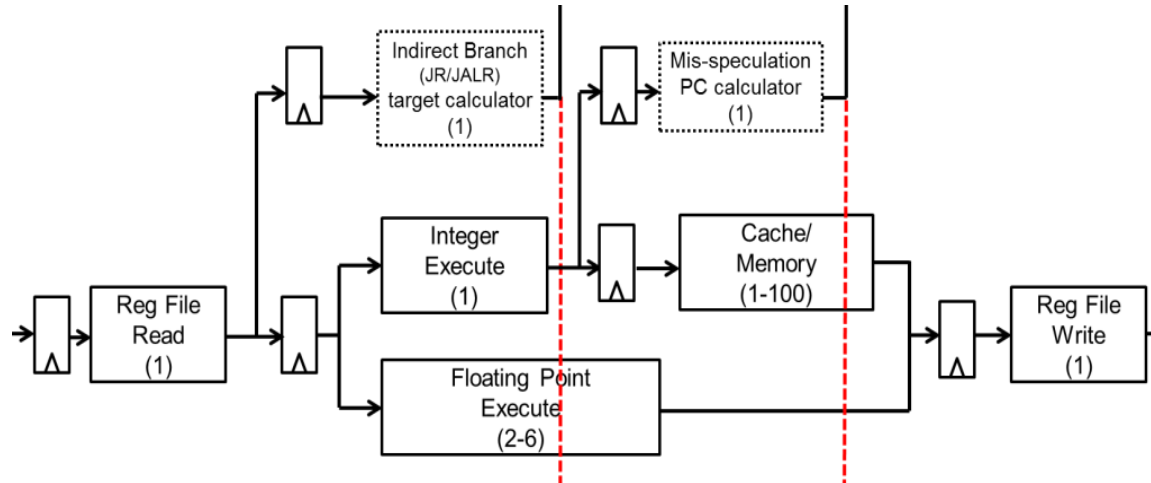


*ROB*

[illegible]

## Ready to execute

# Out-of-Order Execution



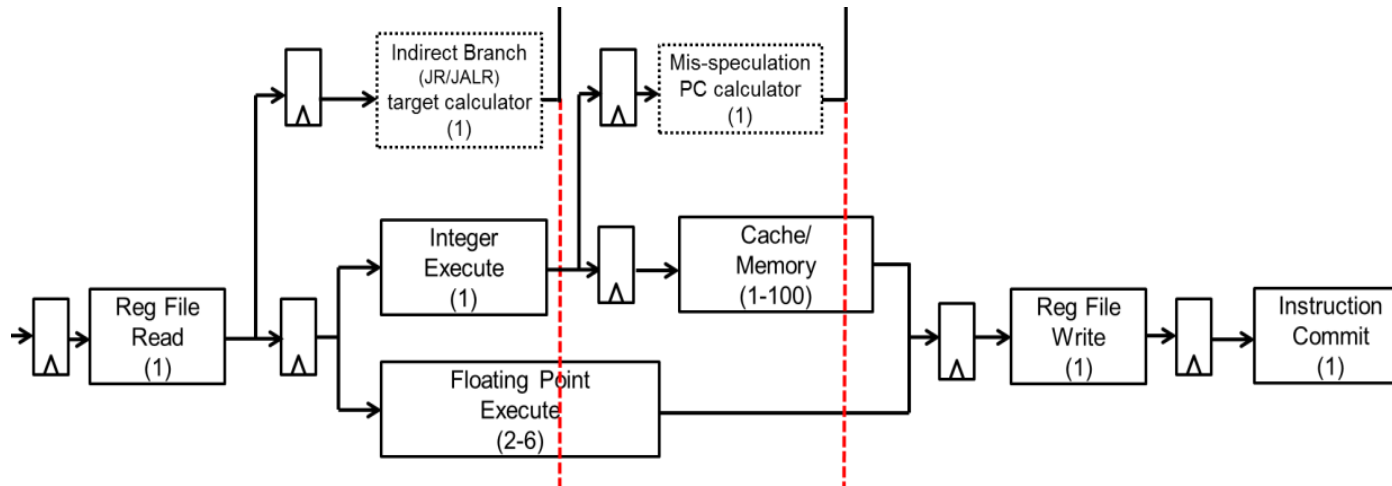
## Reg File Read:

- Operand values are read from the unified physical register file

## Execute:

- Integer and floating point operations are sent to the appropriate functional units
- Stores enter the speculative store buffer
- Loads read from the store buffer or cache/memory

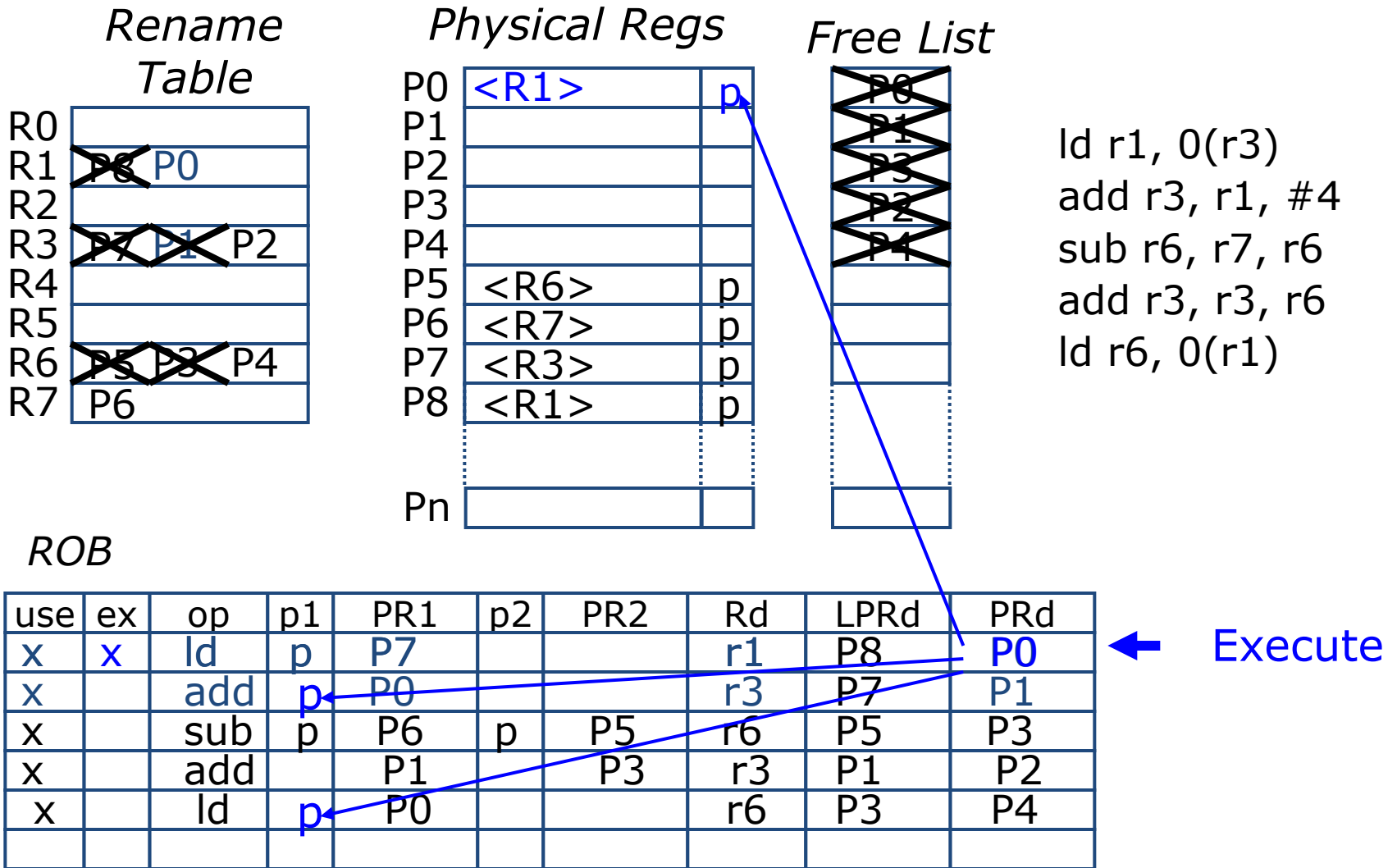
# Out-of-Order Execution



## Reg File Write:

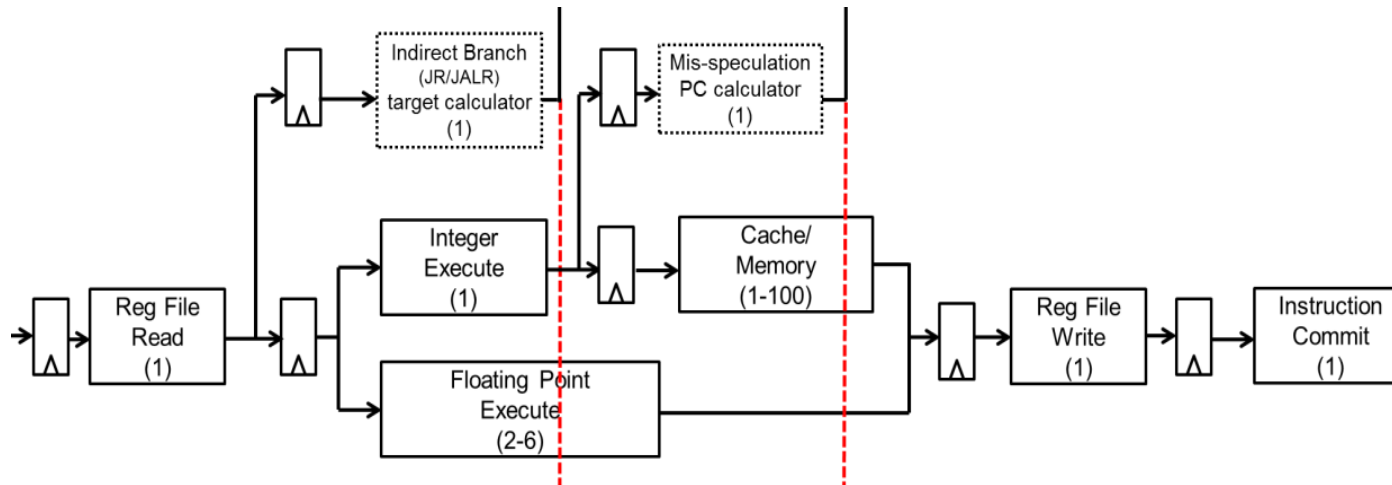
- The output from the functional units/memory is written into the unified register file and the ROB is notified.

# Out-of-Order Execution





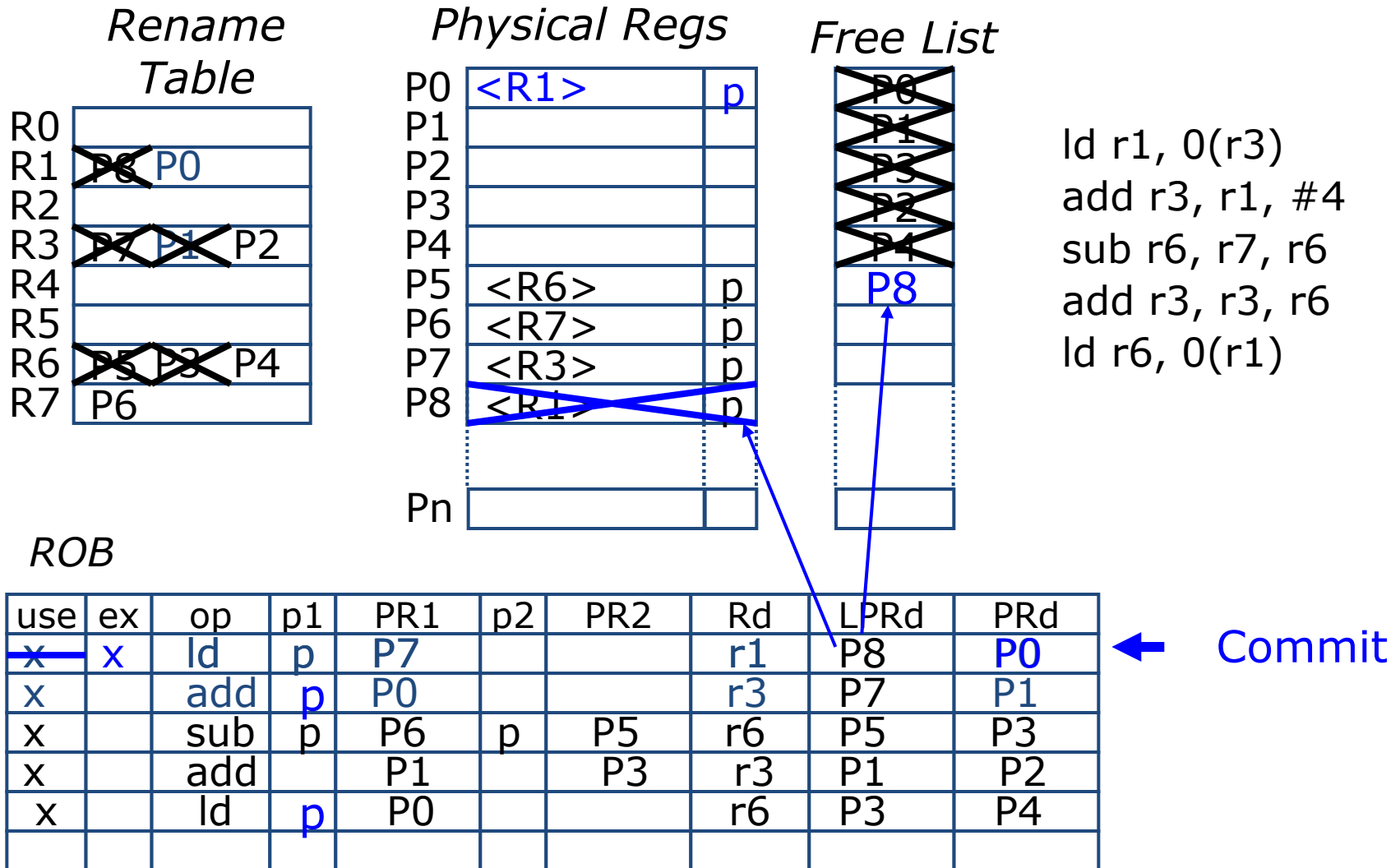
# Out-of-Order Execution



## Commit:

- Instructions are committed in-order
- Free the previously mapped physical register
- Data is written to cache/memory/non-speculative store buffer when a store is committed
- The ROB entry is freed after commit

# Out-of-Order Execution



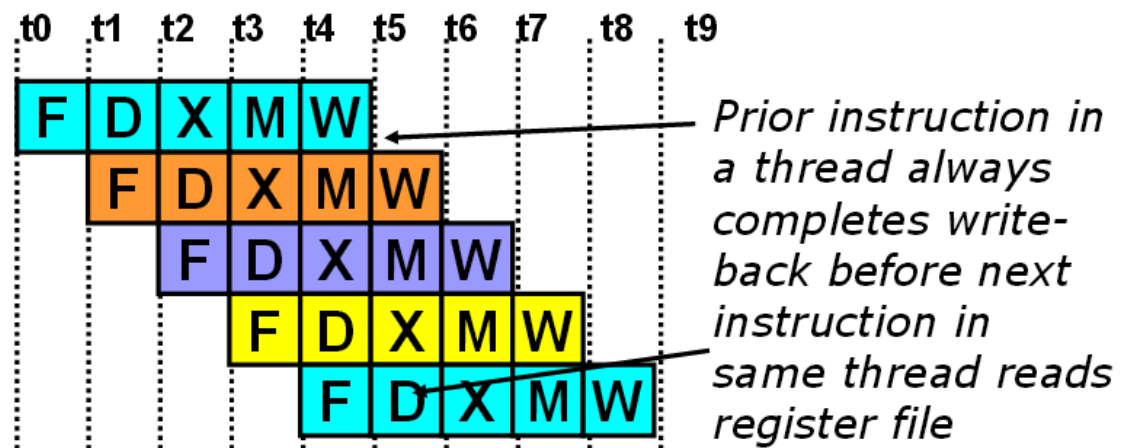
# Multithreading

How can we guarantee no dependencies between instructions in a pipeline?

**Take instructions from different programs**

*Interleave 4 threads, T1-T4, on non-bypassed 5-stage pipe*

T1: LW r1, 0(r2)  
T2: ADD r7, r1, r4  
T3: XORI r5, r4, #12  
T4: SW 0(r7), r5  
T1: LW r5, 12(r1)

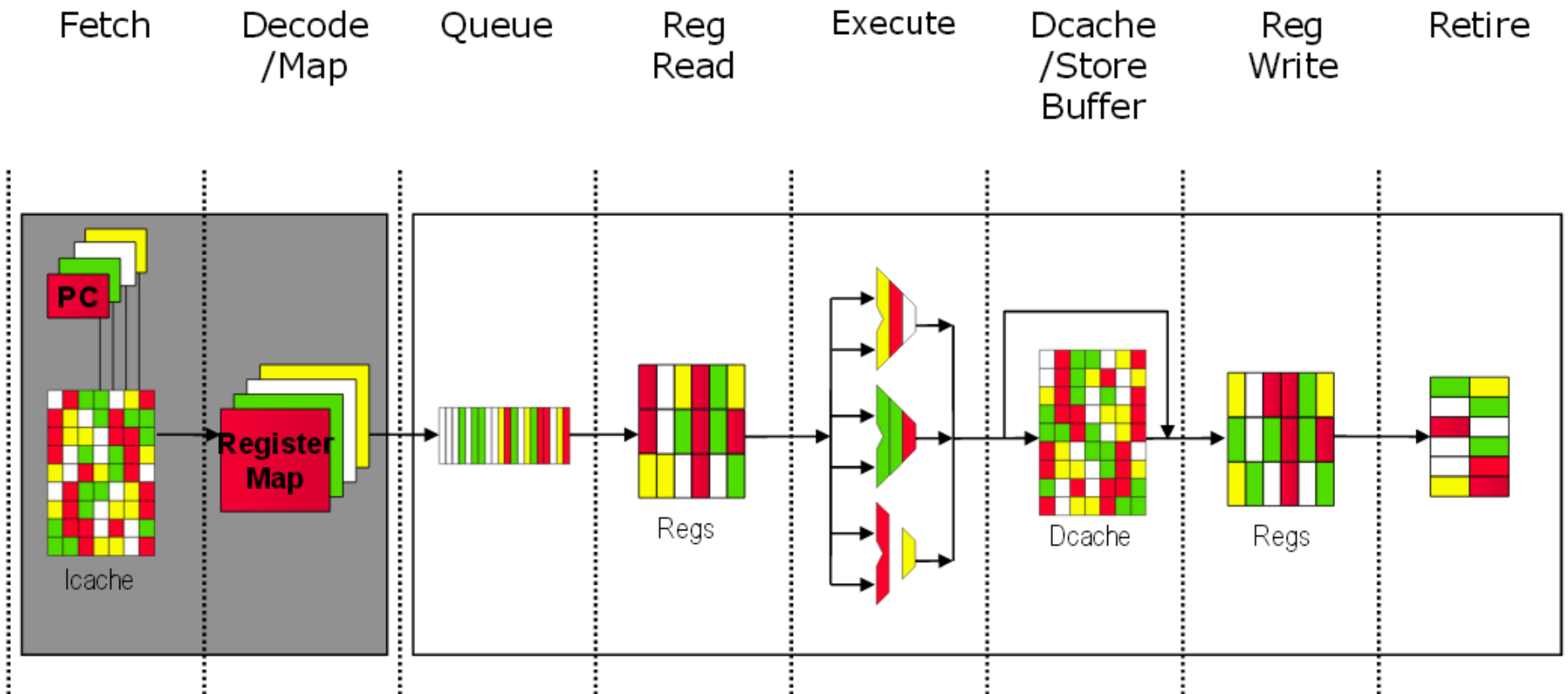


# Scheduling Policy

- Fine-grained multithreading
  - Context switch among threads every cycle
- Coarse-grained multithreading
  - Context switch among threads every few cycles, e.g., on:
    - Function unit data hazard,
    - L1 miss,
    - L2 miss...

# Simultaneous Multithreading (SMT)

- Share OOO structures between threads



# **The end**

Good luck!! 😊