

# Lecture 12:

## Advanced CPU Microarchitecture, pt.1

Jangwoo Kim (Seoul National University)

[jangwoo@snu.ac.kr](mailto:jangwoo@snu.ac.kr)

# Announcement #1

- ◆ HW#3
  - Due: 4/19
- ◆ Mid-term
  - 4/20 (6PM) @ Room 102
- ◆ Warning
  - If you feel **seriously** lost in this class, talk to me.  
Otherwise, think about dropping this course **seriously**.

# Announcement #2

## ◆ Papers to be read

- **Must understand these papers for your mid-term exam!**

*Lec 12* — (1) Superscalar + Out-of-Order execution // the most important one!

- **“The Microarchitecture of Superscalar Processors”** by Smith, Proceedings of IEEE 1995.

*Lec 13* { (2) Simultaneous Multithreading (SMT)  
– **“Simultaneous Multithreading: A Platform for Next-Generation Processors”** by Eggers, et al., IEEE Micro 1997  
(3) Chip Multiprocessor (CMP)  
– **“Niagara: A 32-Way Multithreaded SPARC Processor”** by Kongetira, et al. IEEE Micro 2005

- **What did you learn? Main ideas?**
- **Why important? Any limitations?**

**We DO ask questions based on these readings!!**

# Review: “THREE” Pipeline Hazards

## ◆ Data Hazard

- Data Dependency (Read-after-Write : RAW)
- Anti Dependency (Write-after-Read: WAR)
- Output Dependency (Write-after-Write: WAW)

## ◆ Control Hazard

- Data Dependency of Program Counter

## ◆ Structural Hazard

- Due to lack of resources  
(e.g., # ALU < # instructions ready?)

# Can we make a faster CPU?

# Performance Factors

$$T_{\text{wall-clock}} = T_{\text{cyc}} \times \text{CPI} \times \text{No. Instructions}$$

max. combinational delay



cycles-per-instruction



ISA and  
compilers



# ILP: Instruction-Level Parallelism

- ◆ ILP is a measure of the amount of inter-dependencies between instructions

- ◆ Average ILP = # of instructions / # of cyc required

code1:     ILP = 1

i.e. must execute serially

code2:     ILP = 3

i.e. can execute at the same time

code1:	$r1 \leftarrow r2 + 1$
	$r3 \leftarrow r1 / 17$
	$r4 \leftarrow r0 - r3$

code2:	$r1 \leftarrow r2 + 1$
	$r3 \leftarrow r9 / 17$
	$r4 \leftarrow r0 - r10$

# 5-stage MIPS Machine

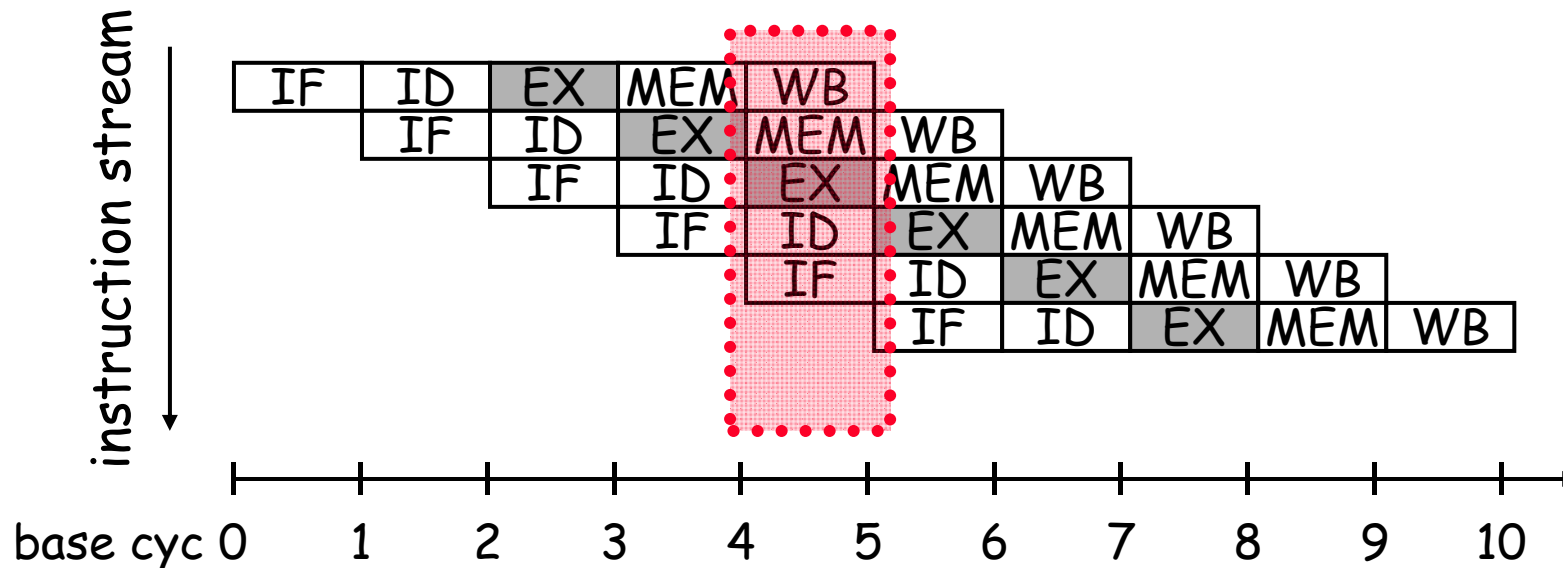
Scalar Pipeline (baseline)

Operation Latency = 1

Peak IPC = 1

Instruction-Level Parallelism (ILP)

= # of instruction / # of cycles required = 1





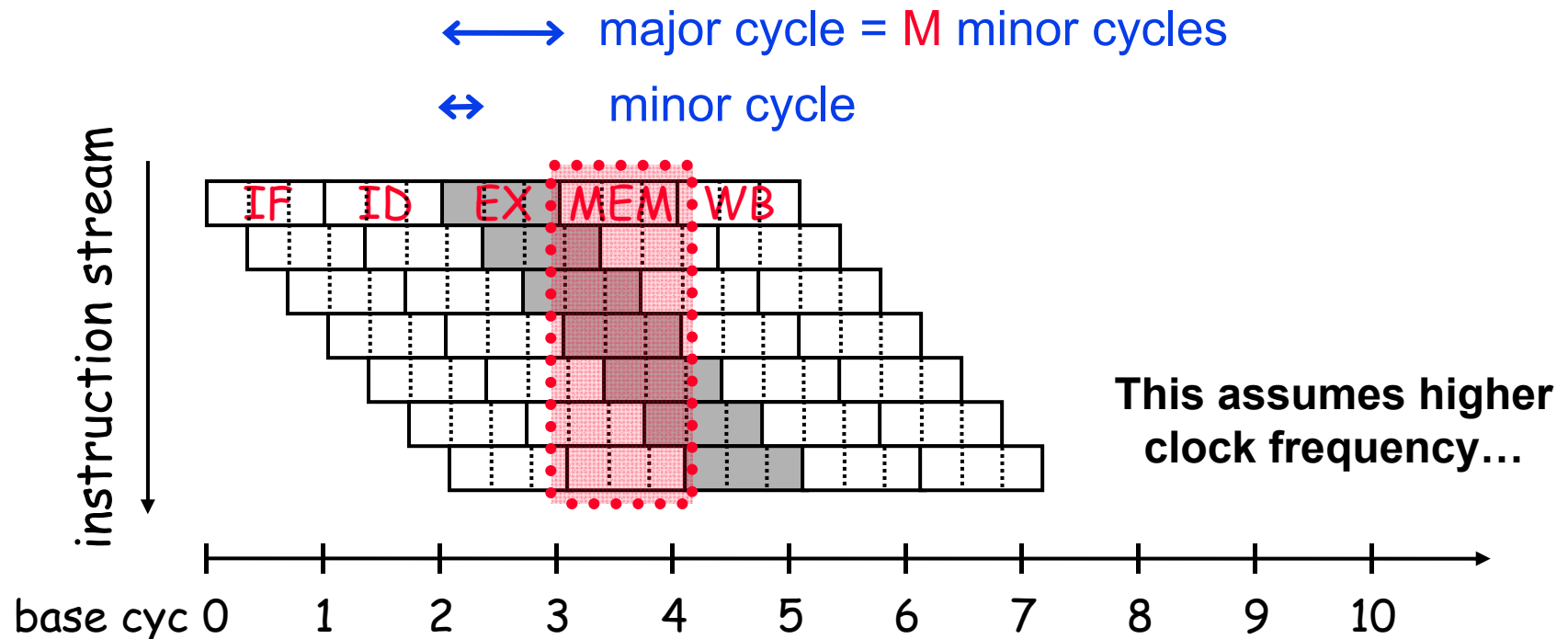
# Superpipelined Machine

## Superpipelined Execution

OL = 1 baseline cycle (M minor cycles)

Peak IPC = M per baseline cycle (1 per minor cycle)

ILP = M



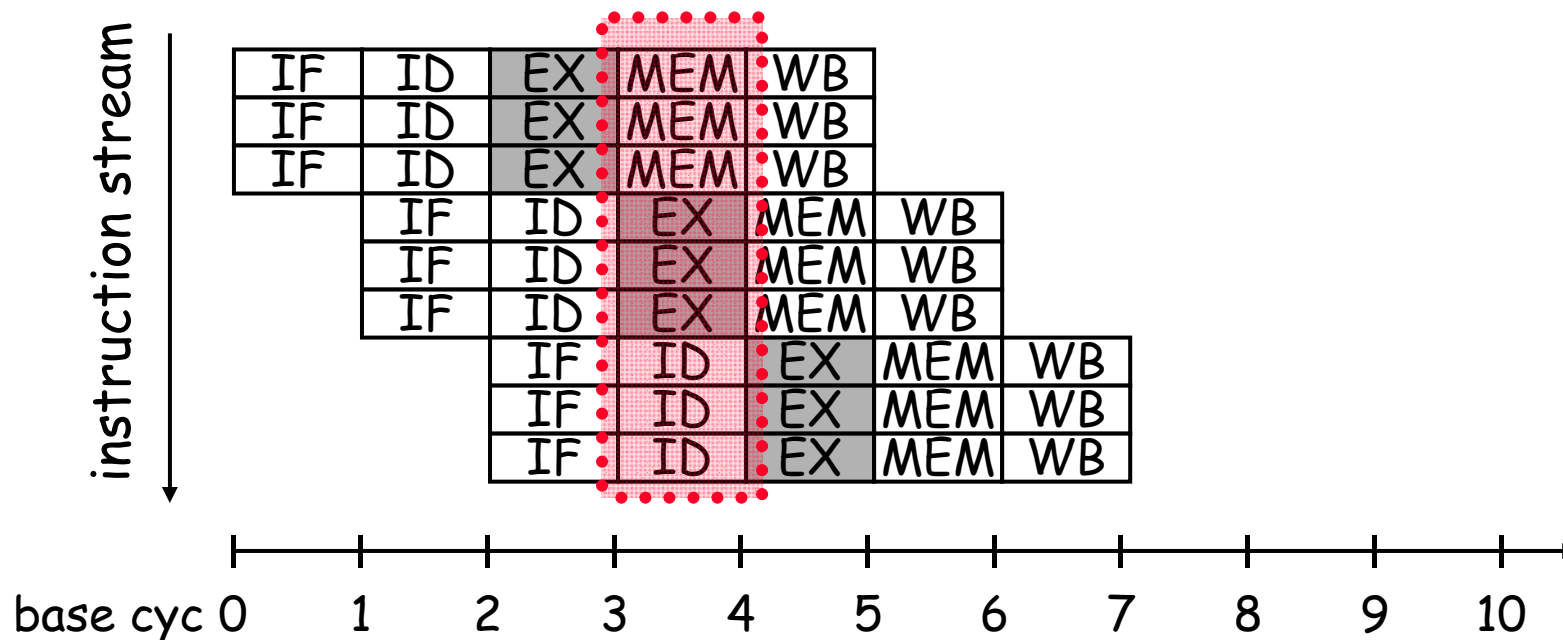
# Superscalar Machines

## Superscalar (Pipelined) Execution

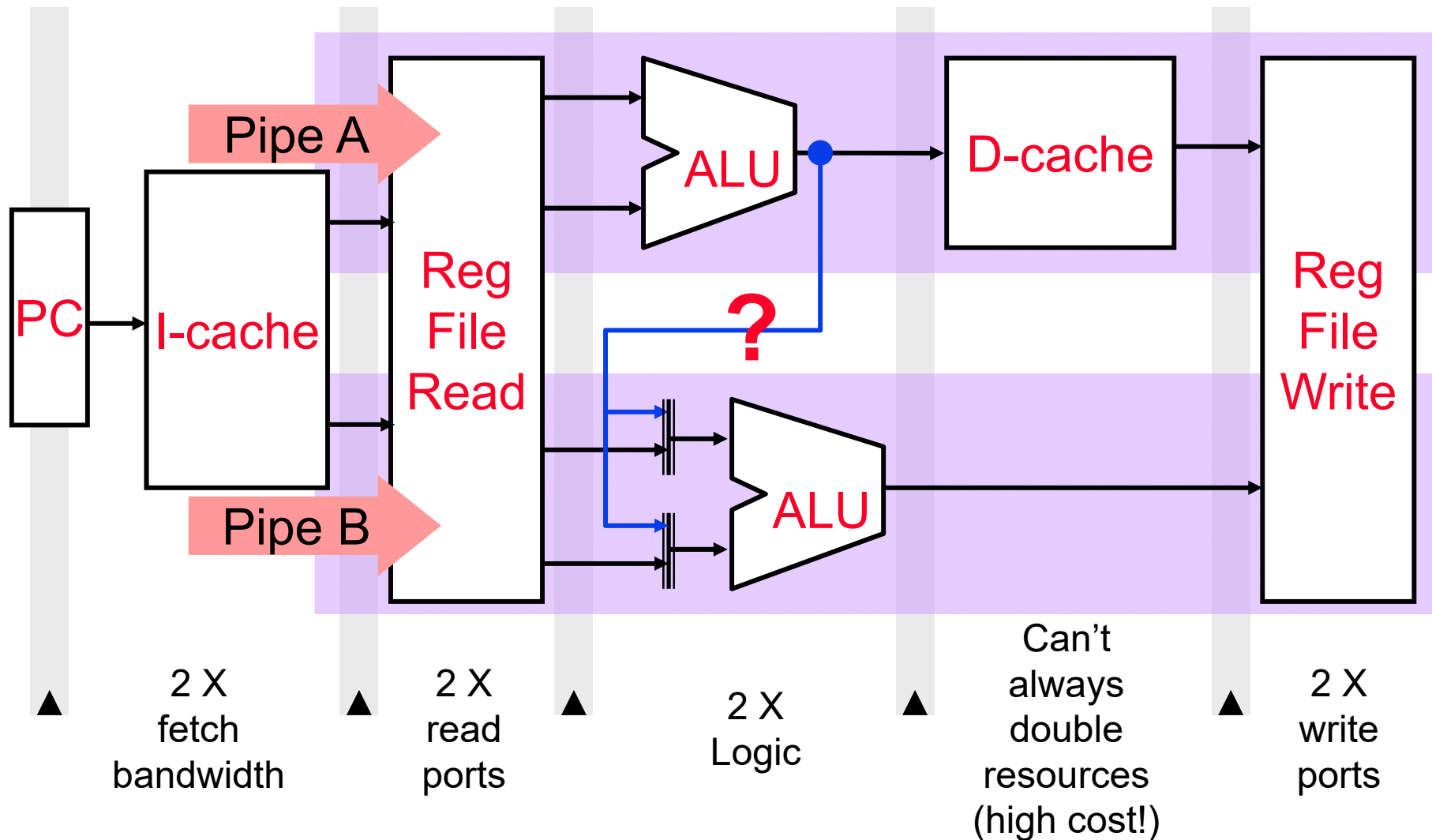
OL = 1 baseline cycle

Peak IPC = N per baseline cycle

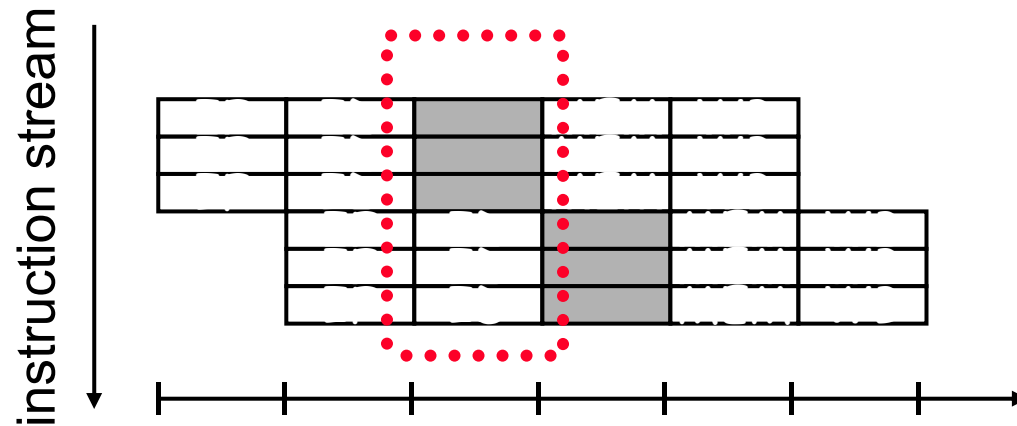
ILP = N



# 'Example' 2-way Superscalar Datapath



# Superscalar and Superpipelined



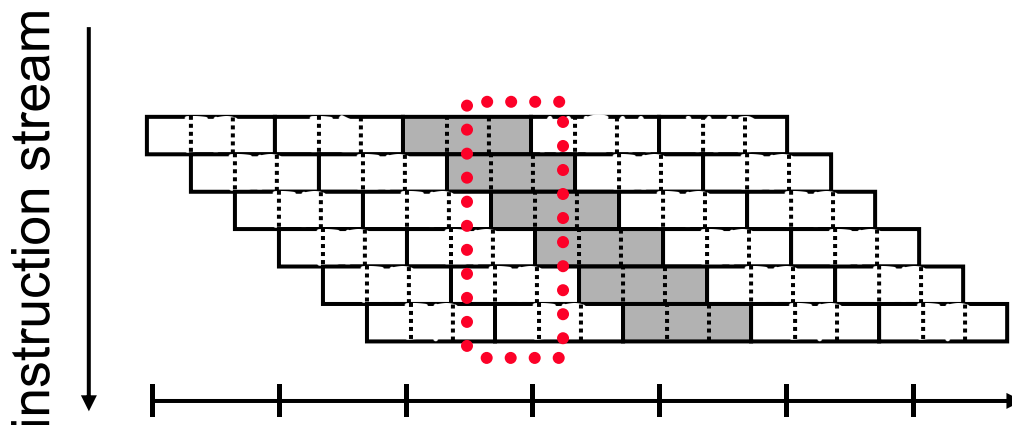
## Superscalar Parallelism

Operation Latency: 1

Issuing Rate:  $N$

Superscalar Degree:  $N$

VS.



## Superpipeline Parallelism

Operation Latency: 1

Issuing Rate:  $M$

Superpipelined Degree:  $M$

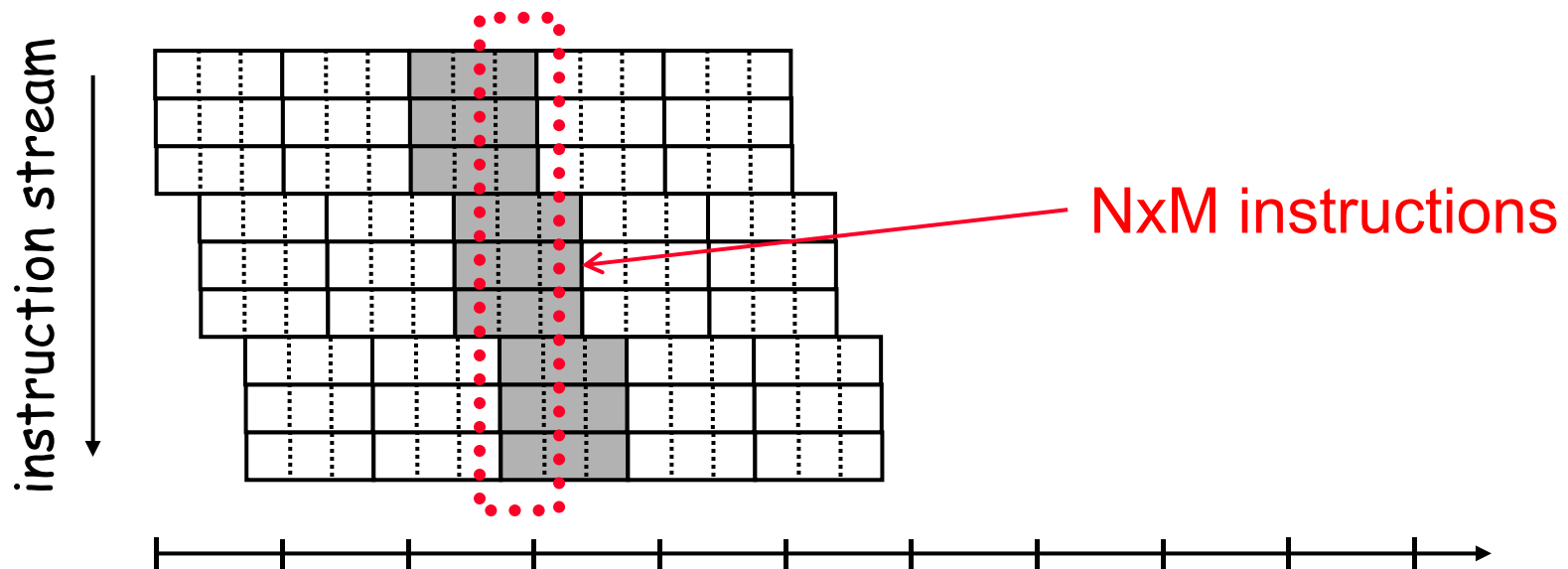
Achieving peak performance on each architecture depends on finding  $N$  or  $M$  independent instructions per cycle

# Limitations of In-Order Pipelines (1/2)

- ◆ **IPC of In-Order pipelines degrades very sharply** if the machine parallelism is increased beyond a certain point, **i.e. when  $N \times M$  becomes similar to the average distance between dependent instructions**

- ◆ **Forwarding is no longer effective**

**Even with full forwarding, pipeline may never be full due to frequent dependency stalls!!**



# Limitations of In-Order Pipelines (2/2)

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
<code>addf f2,f0,f1</code>	F	D	E+	E+	E+	W										
<code>mulf f3,f2,f4</code>		F	D	d*	d*	E*	E*	E*	E*	E*	W					
<code>subf f5,f6,f7</code>			F	p*	p*	D	E+	E+	E+	W						

What's happening in cycle 4?

- `mulf` stalls due to **RAW hazard**
  - OK, this is a fundamental problem
- `subf` stalls due to **Pipeline hazard**
  - Why? `subf` can't proceed into D because `mulf` is there
  - This is NOT a fundamental problem.


Why can't `subf` go into D in cycle 4 and E+ in cycle 5?

But, if we allow this, there might be other problems.....

# What is TRUE data hazard?

## Data dependence

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




## True dependency

Read-after-Write  
(RAW)

## Anti-dependence


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



Write-after-Read  
(WAR)

## Output-dependence

$r_3 \leftarrow r_1 \text{ op } r_2$   
 $r_5 \leftarrow r_3 \text{ op } r_4$   
 $r_3 \leftarrow r_6 \text{ op } r_7$



## False dependency

Write-after-Write  
(WAW)

# What is TRUE data hazard?

Data dependence

Can't do anything for this

$r_3$  op  $r_4$

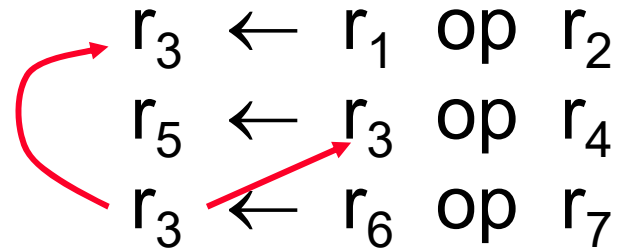
Read after-Write  
(RAW)

Can remove this  
if we have more resources



# Removing False Dependencies

- ◆ Anti (WAR) and output (WAW) dependencies are “false” dependencies



```
r3 ← r1 op r2
r5 ← r3 op r4
r3 ← r6 op r7
```

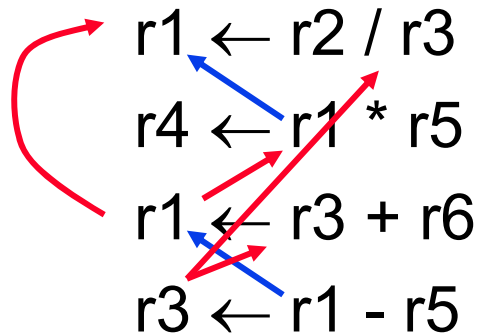
- ◆ The dependence is just on the register “name” rather than “data.” **What if we could use more registers?**
- ◆ Given **infinite** number of registers, anti (WAR) and output (WAW) dependencies can be eliminated completely.

# Register Renaming: Example

Original

```

r1 ← r2 / r3
r4 ← r1 * r5
r1 ← r3 + r6
r3 ← r1 - r5
  
```

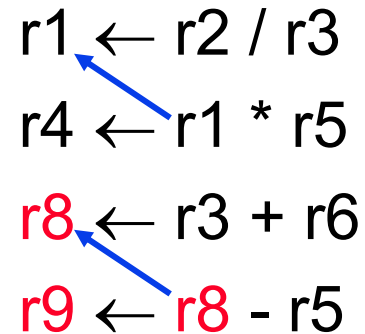
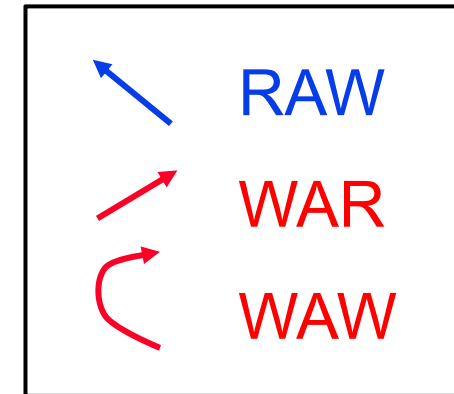


ILP=1

Renamed

```

r1 ← r2 / r3
r4 ← r1 * r5
r8 ← r3 + r6
r9 ← r8 - r5
  
```

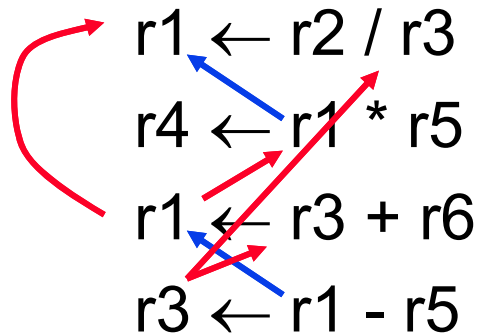



# Register Renaming: Example

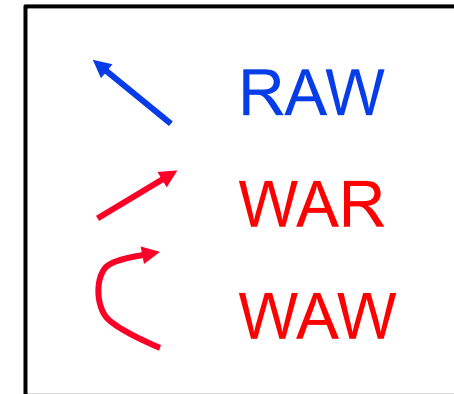
Original

```

r1 ← r2 / r3
r4 ← r1 * r5
r1 ← r3 + r6
r3 ← r1 - r5
  
```



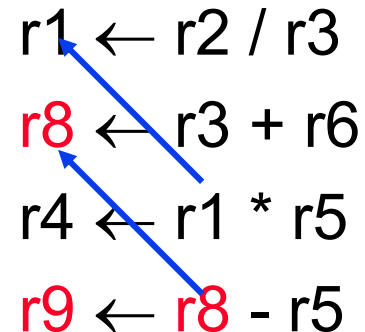
ILP=1



Renamed

```

r1 ← r2 / r3
r8 ← r3 + r6
r4 ← r1 * r5
r9 ← r8 - r5
  
```



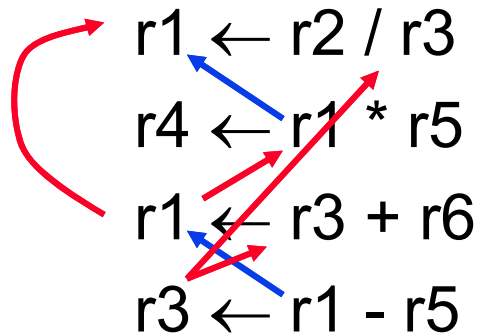
ILP=2 (superpipelining)  
with changed ex order

# Register Renaming: Example

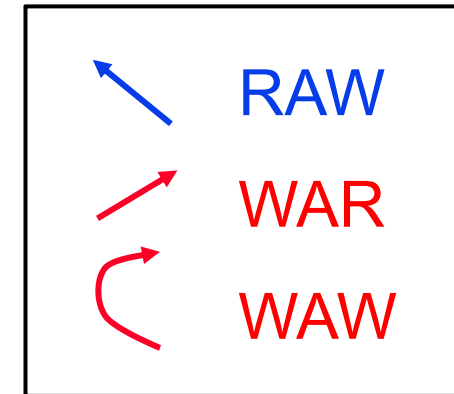
Original

```

r1 ← r2 / r3
r4 ← r1 * r5
r1 ← r3 + r6
r3 ← r1 - r5
  
```



ILP=1



Renamed

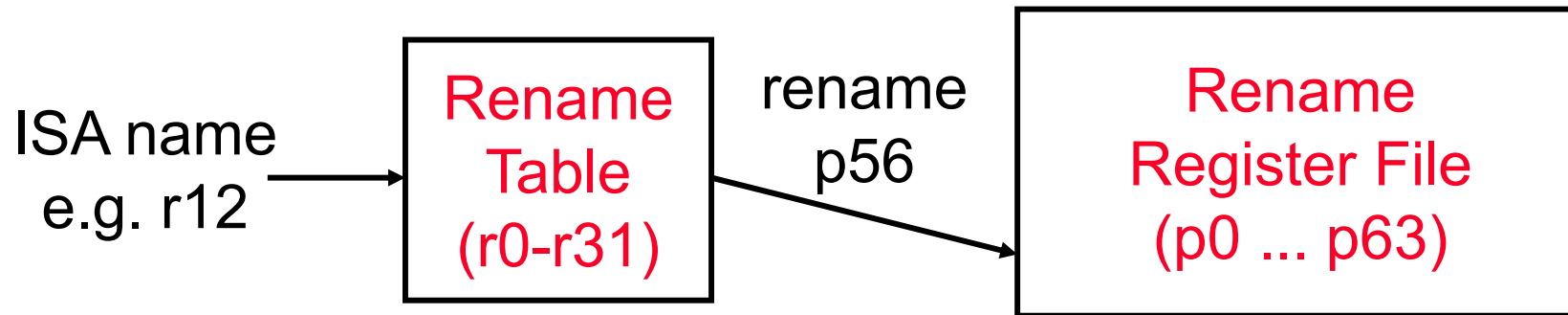
```

r1 ← r2 / r3
r4 ← r1 * r5
r8 ← r3 + r6
r9 ← r8 - r5
  
```



ILP=2 (superscalar)  
with multiple execution

# Hardware Register Renaming (1/2)



- ◆ Renaming maintains bindings from ISA reg. names to uArch. reg. names
  - Compiler does not know about uArch. rename registers.
- ◆ When issuing an instruction that updates 'architecture' register 'rd':
  - Allocate an unused rename 'physical' register 'px'
  - Record binding from 'rd' to 'px'
- ◆ When to remove a binding? When to de-allocate a rename register?

# Hardware Register Renaming (2/2)

- ◆ Anti (**WAR**) and Output (**WAW**) dependencies are false
  - The dependence is on name/location rather than data
  - Given infinite registers, WAR/WAW can always be eliminated
  - Renaming removes these dependencies, but **RAW** remains.
- ◆ Example
  - Names (=logical/arch registers) : **r1, r2, r3**
  - Locations (=physical registers) : **p1, p2, p3, p4, p5, p6, p7**
  - Original mapping: **r1→p1, r2→p2, r3→p3** and **p4~p7 = “free”**

MapTable

r1	r2	r3
p1	p2	p3
p4	p2	p3
p4	p2	p5
p4	p2	p6

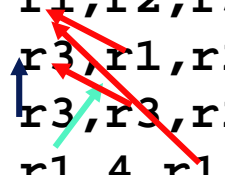
FreeList

p4, p5, p6, p7
p5, p6, p7
p6, p7
p7

Orig. insns

```

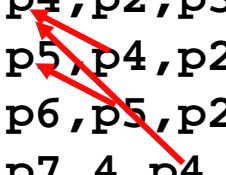
add r1, r2, r3
sub r3, r1, r2
mul r3, r3, r2
div r1, 4, r1
  
```



Renamed insns

```

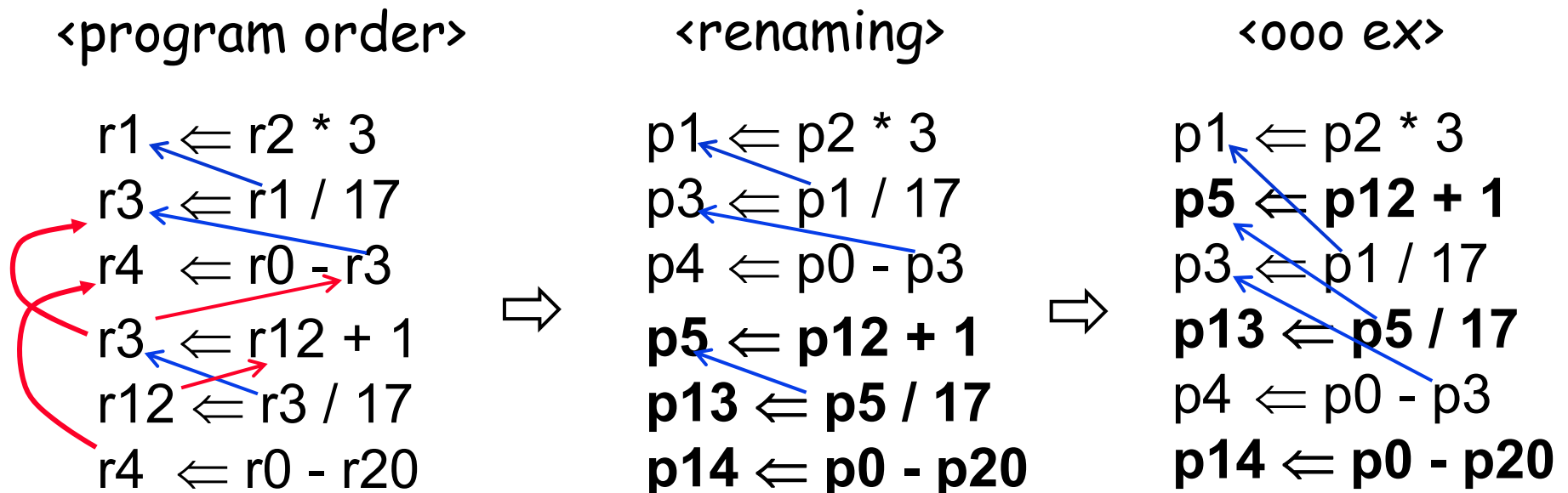
add p4, p2, p3
sub p5, p4, p2
mul p6, p5, p2
div p7, 4, p4
  
```



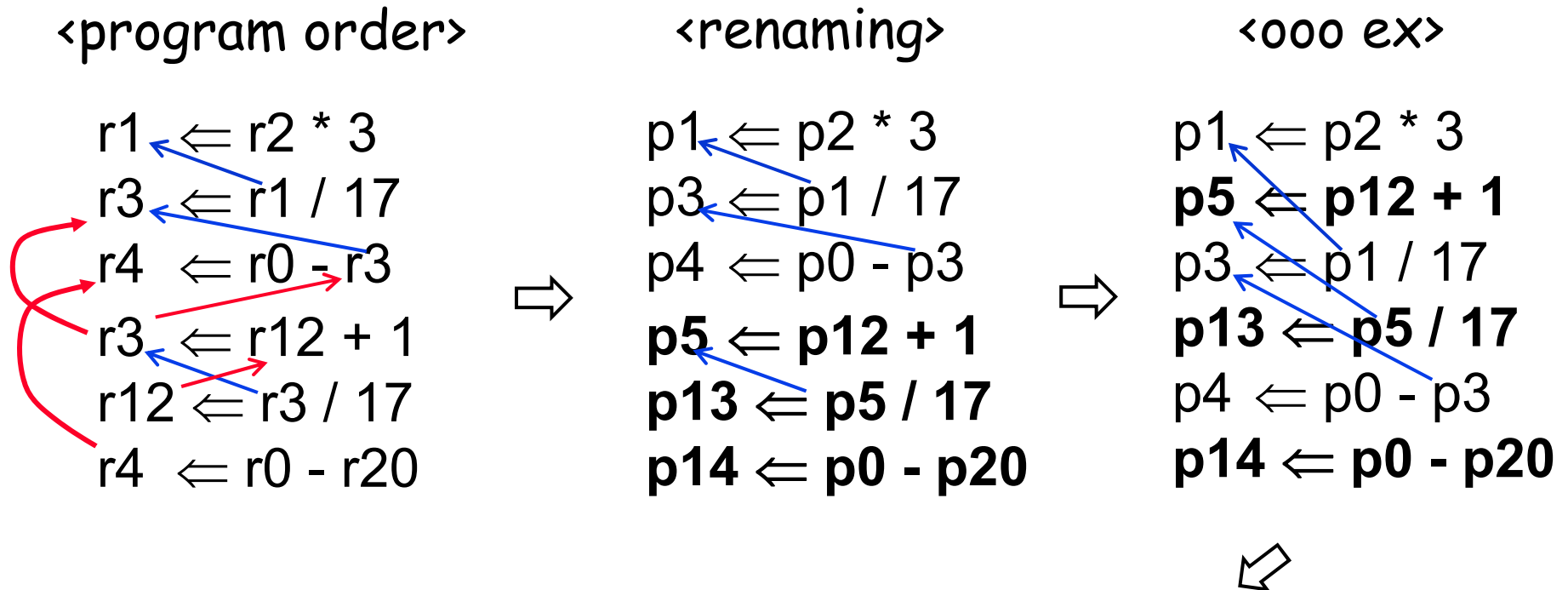
# Out-of-Order Execution (1/2)

- (1) “Register Renaming” effectively eliminates WAW and WAR
- (2) In a RAW dependent instruction pair, the reader must wait for the result from the writer
- (3) Otherwise, any non-dependent instructions can start their own executions → OoO execution!

note: “issue” = send instructions to Ex/Mem units



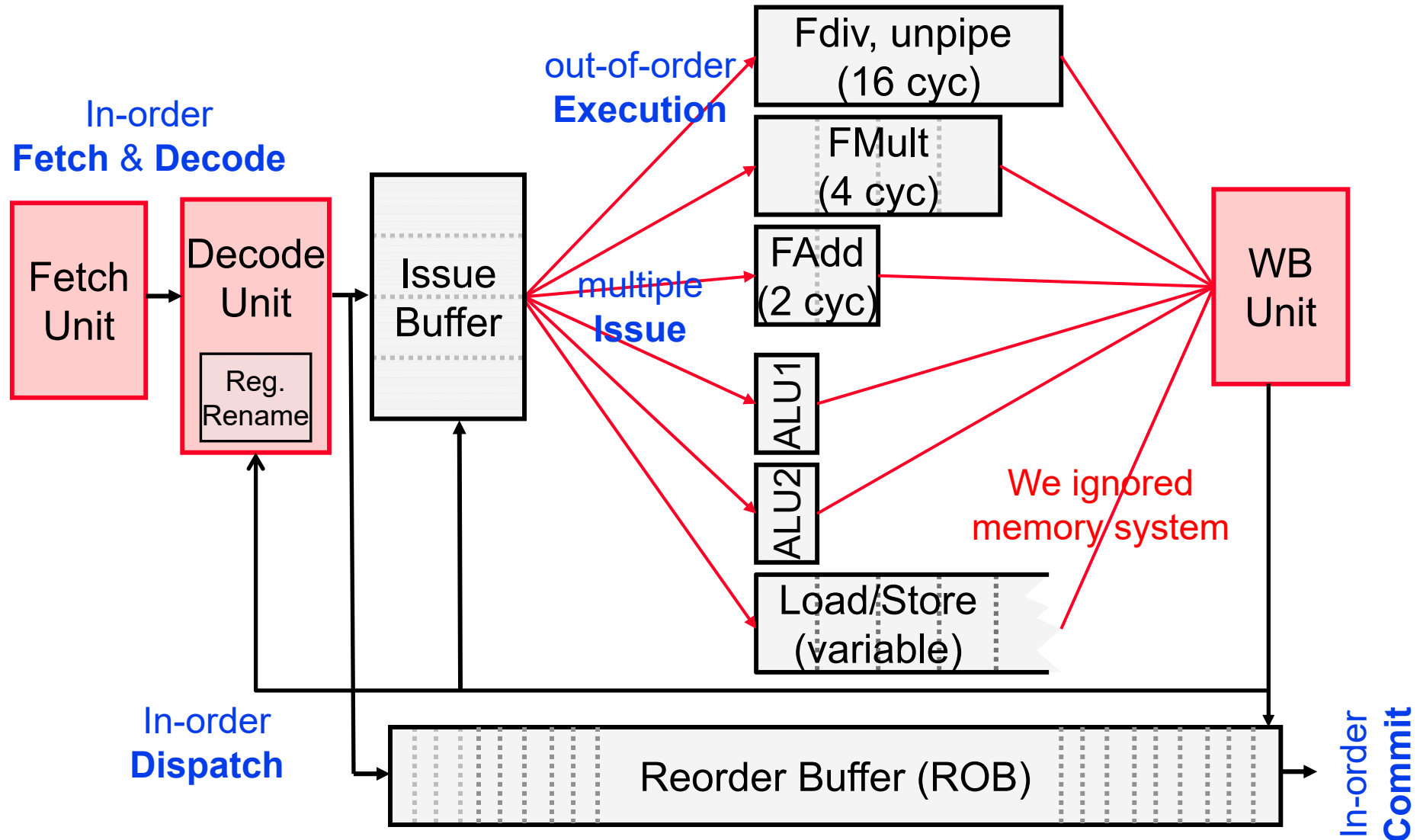
## Out-of-Order Execution (2/2)



- If dep. distance > issue distance, even RAW is eliminated  
(at least for performance impact)
- With OoO, **superscalar** gets much better!  
(easy to fill multiple pipelines (=high ILP))
- But, OoO must not be exposed to the outside of CPU.



# Typical Out-of-Order Superscalar CPU



The Microarchitecture of Superscalar Processors [Smith 1995, Proceedings of IEEE]

# Dataflow Execution Ordering

- ◆ Maintain an window of many pending instructions (a.k.a. '**Issue Buffer**' or '**Issue Queue**')
  - ◆ Issue instructions out-of-order
    - Find instructions whose operands are ready
    - **Issue Queue (ISQ)** must check if the outcomes of newly completed instructions match operands of pending instructions in ISQ to enable timely OoO issue without violating **RAW** dependency.
    - Issue the older instructions among ready instructions
- ◆ Must still remember the original program order.
  - **Why?**
    - Programmer assumes instructions completing in program order
    - Branch prediction, precise interrupt, ...
  - **Reorder Buffer (ROB)** remembers the program order.

# Instruction Reorder Buffer (ROB)

- ◆ At today's clock frequency, on a memory load
  - a cache hit (best case) takes 4~7 cyc
  - a L1 cache miss takes a few 10s of cycles
  - an off-chip cache miss takes a few 100s of cycles→ A lot of following instructions cannot leave the pipeline!
  
- ◆ **ROB** is a program-order instruction bookkeeping structure
  - Instructions must enter and leave in program order
  - : **Programmer's View!**
  - Holds 10s to 100s of "in-flight" instructions in various stages of execution
  - Re-sorts all instructions on exit to appear to complete in program order
  - Supports 'precise & transparent' exception for any in-flight instruction

# Load/Store Queue (LSQ)

- ◆ ROB makes register writes in-order, but **what about stores?**
- ◆ Will you allow instructions in the pipeline to change DM (data memory) in execution stage?
  - Not even close, imprecise memory worse than imprecise registers
- ◆ **Load/store queue (LSQ)**
  - Completed stores write the values to LSQ
  - When store retires, the head of LSQ is written to DM
  - When loads execute, access LSQ and DM in parallel
    - Forward values from LSQ if the address matches.
    - More modern design: loads and stores in separate queues
  - More on this later

# Control Dependency can hurt ILP

Suppose we have an infinitely wide datapath, perfect renaming and an infinitely large issue buffer, what is the still critical problem remaining?

- ◆ Control-transfer instructions (e.g., branches and jumps) occupy 14% of an avg. instruction mix.

How good branch prediction necessary?

e.g., ~ 18 branches in 128 in-flight instructions

90% correct branch prediction → 15% of filling ISQ/ROB

95% → 40%, 97% → 58%, 99% → 83%, ...

Then, how can we keep Issue Queue & ROB filled?

# Branch Prediction in SS/OoO

- ◆ Superscalar + OoO execution
  - Many in-flight instructions necessary to keep CPU busy
  - Require extremely accurate branch prediction
  - Modern techniques get better than 95% accuracy
  
- ◆ The penalty of wrong-path prediction increases
  - Increasing # of wrong-path instructions until the outcome of predicted branch is known
  - **Recovery from a wrong prediction gets very complicated**
    - How to restore **the original value of registers** that wrong-path instructions might have updated?
    - How to restore **renaming table & free registers**?
    - Speculative branch prediction table updates (and restore if wrong?)

# Speculative Execution in SS/OoO

- ◆ Instructions after a predicted branch are speculative
  - Need a mechanism to undo their effects!!
- ◆ Must be able to maintain separate copies of
  - **In-Order State**: a check-point state up to just before the first speculated instruction
  - **Speculative State**: include all state changes after check-point, possibly multiple predicted branches

- ◆ **Commit**

Admit known-to-be good speculative state changes into the in-order state  
= Remove the oldest, completed instructions from the pipeline = Remove completed instructions from the head of ROB

- ◆ Rewind - discard all, or part of, the speculative state

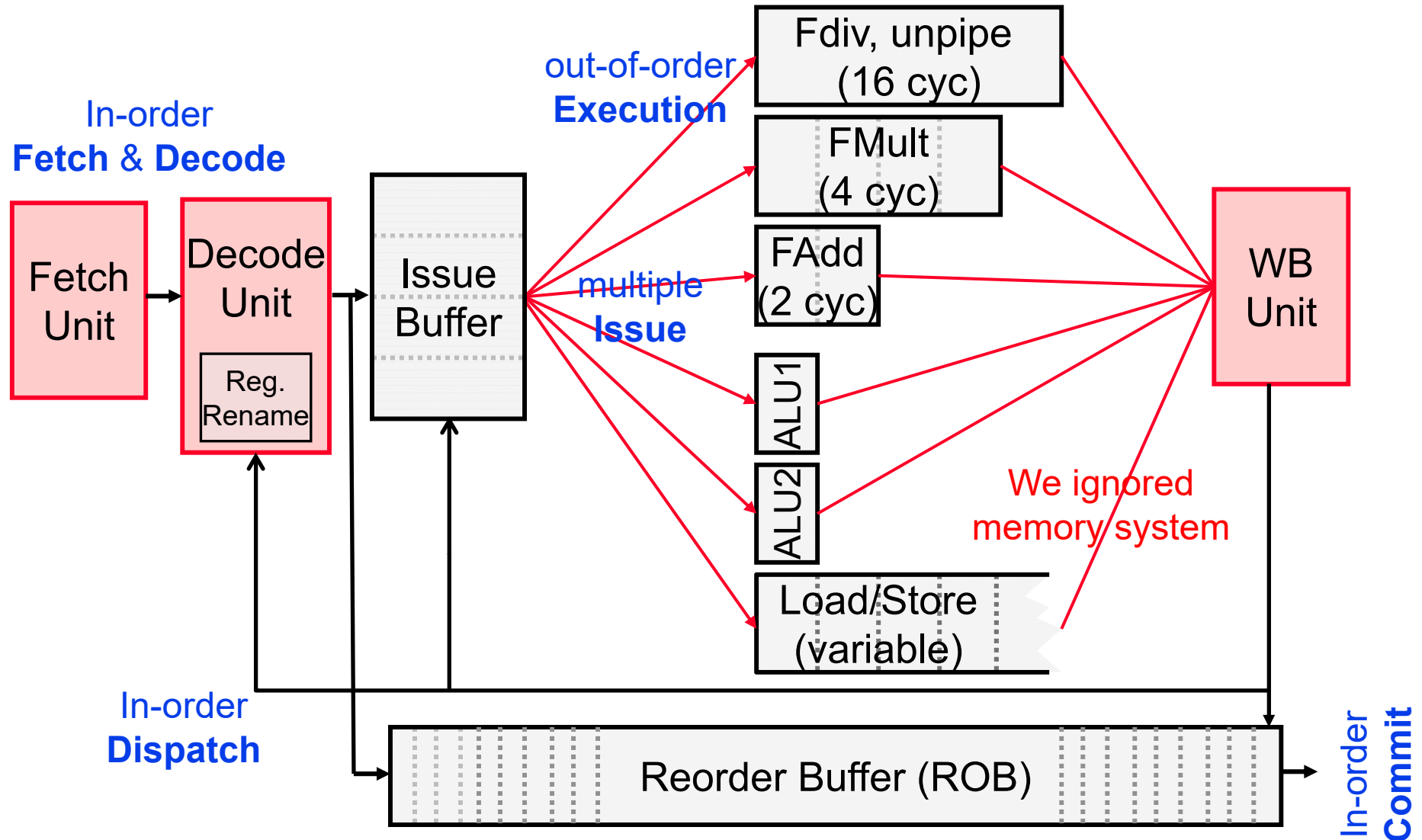
Actually, a lot of non-branch instructions can be also speculative instructions. Why?

# Superscalar + Out-of-Order Execution

- ◆ In-order Fetch
- ◆ In-order Decode
  - Register Renaming
- ◆ Out-of-Order issue & complete
  - Issued from Issue Queue & Load-Store queue
  - Operand obtained from
    - Arch. register (=logical register)
    - Physical register (or ROB entries) // two ways of reg. renaming
  - Update physical registers (or ROB entries)
- ◆ Out-of-Order complete
- ◆ In-order commit
  - Physical register written to architecture register
  - Programmer's view of sequential & atomic inst execution



# Typical Out-of-Order Superscalar CPU



Read "The Microarchitecture of Superscalar Processors" [Smith 1995, Proceedings of IEEE]

# Going Forward: What's Next

## ◆ History

- “Scoreboarding” - First OoO without register renaming
- “Tomasulo’s algorithm” - adds register renaming

## ◆ Modern OoO mechanisms

- Handling precise state and speculation
  - **P6-style execution (Intel Pentium Pro)**
    - ROB works like internal register file + issue queue
    - Separate “architecture registers” for program visible state
  - **R10k-style execution (MIPS R10000)**
    - ROB remembers only the program order (not value).
    - Big physical register files (implicitly include arch. RF)
- Handling memory dependencies in more advanced ways
- Read “**The Microarchitecture of Superscalar Processors**”

**We will cover more details in grad-level courses**

Advanced Computer Architecture  
Parallel Computer Architecture

# Register Renaming in P6 and R10K

## P6

- Map table: **Register** → “**Arch. register**” or “**ROB entry**”
- Completed value written to ROB entry indicated by map table
  - ROB works like temporal registers by storing ‘values’
  - ROB entry has a bit to tell whether result is pending or completed
- The oldest ‘completed’ instructions written to a separate **Arch. Register file** (= program visible state.)
- No free list required because the tail of ROB is the first free register.
- ROB is not scalable

## R10K

- **Physical register file** holds all values → **No Arch. Register file**
- # physical registers = # architectural registers + # temporal registers
- Map all architectural registers to physical registers
- Arch. reg file can be “extracted” from physical reg. file
- Free list must keep a track of unallocated physical registers

# Freeing Registers in P6 and R10K

## P6

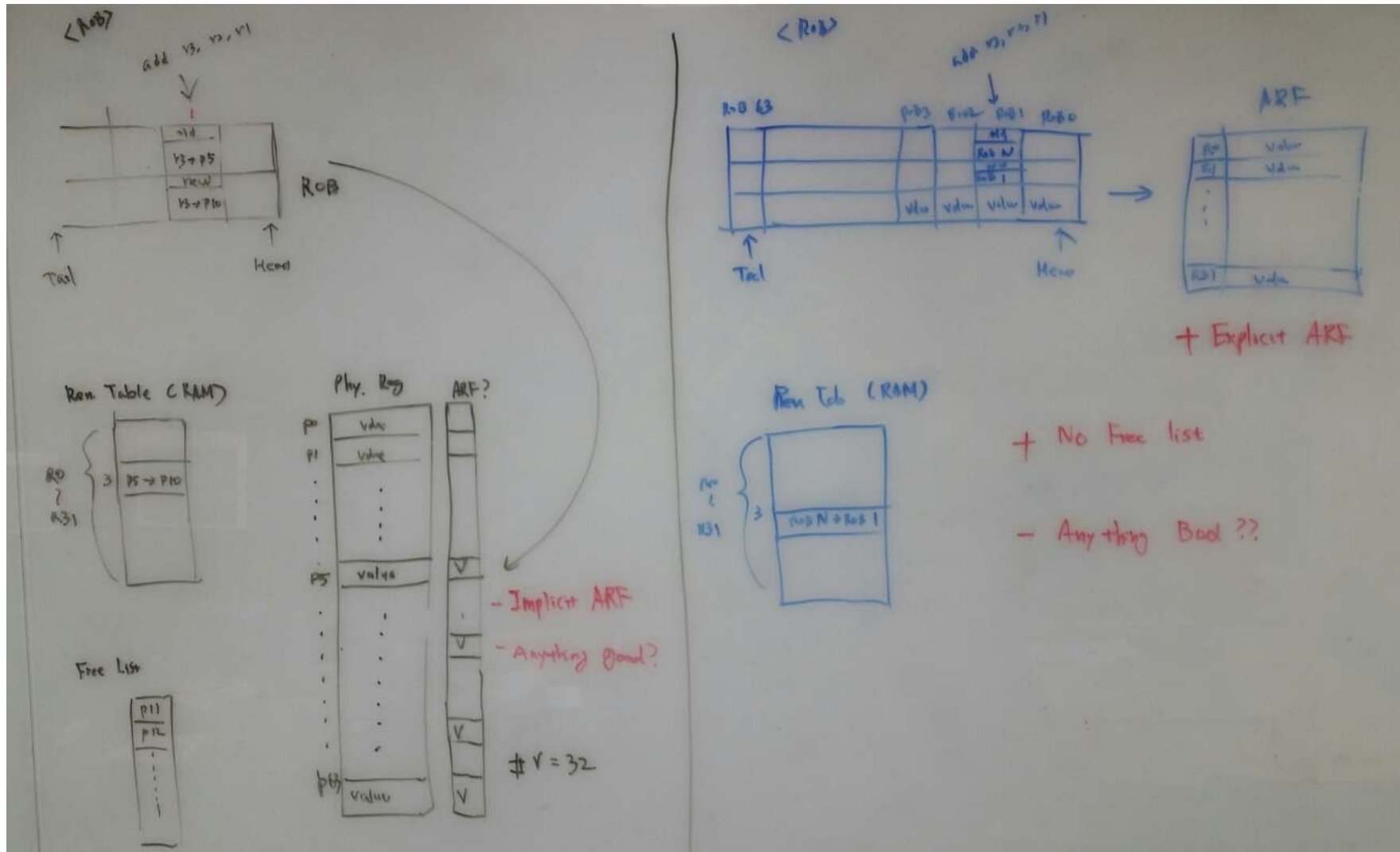
- No need to free storage for speculative (“in-flight”) values explicitly
  - Temporary storage comes with an ROB entry
- When an instruction is retired (=commit), copy speculative value from the head of ROB to Arch. register file (ARF) and free the ROB entry
  - So, ARF is always the program visible register file.

## R10K

- Just of a large PRF here. Then, how can you extract its Arch. register file?
- Can't free a physical register even after its renamed instruction is retired from the pipeline.
- Only can free a physical register previously mapped to the same logical register becomes the oldest instruction in the pipeline
  - All insnts that will ever read its value must have retired!



# R10K style vs. P6 style

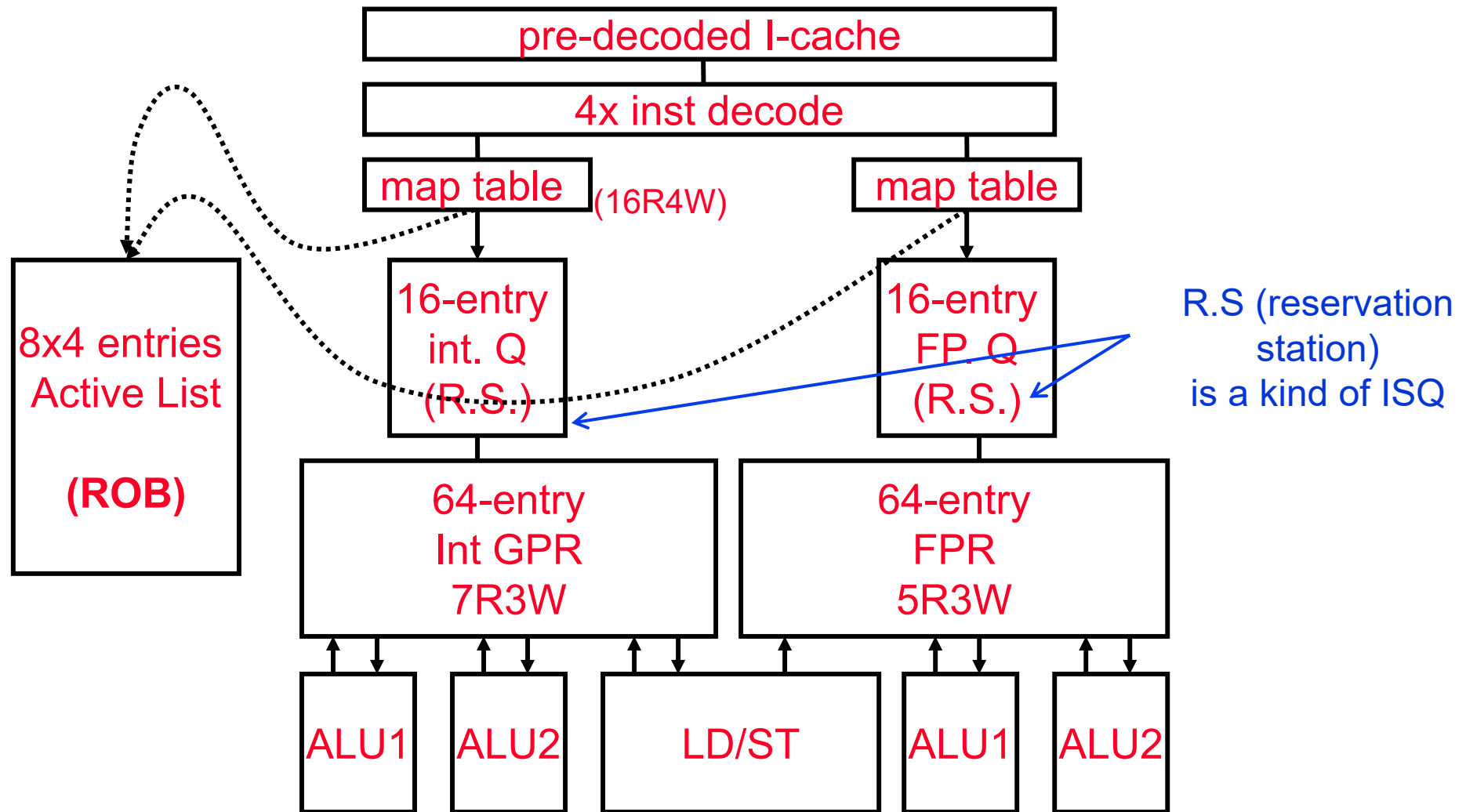


- + Scalable ROB (no value)
- Free list management

- Complicated ROB (read ports)
- Multiple sources of data read

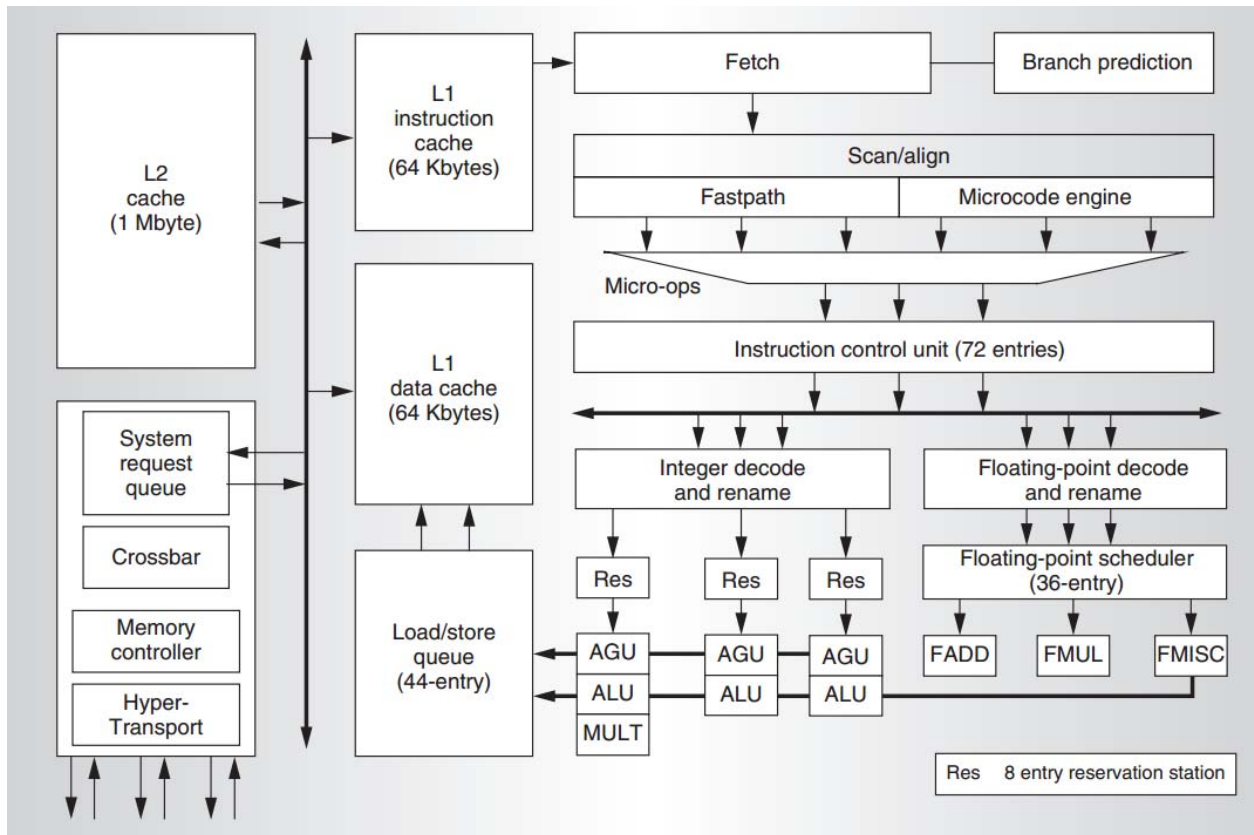
# MIPS R10000

## “Out-of-Order Superscalar CPU”



Read [Yeager 1996, IEEE Micro] if you are really interested

# AMD Opteron “core”



- **x86-64** → RISC ops
  - > 72-entry ROB
  - > 16-entry Arch. Regs
- Register Renaming
  - > 16 arch → 72-entry ROB
  - “value stored in ROB”

**P6 style**
- Out-of-Order
  - > 3 x 8-entry Int ISQ
  - > 36-entry FP ISQ
  - > 44-entry LSQ

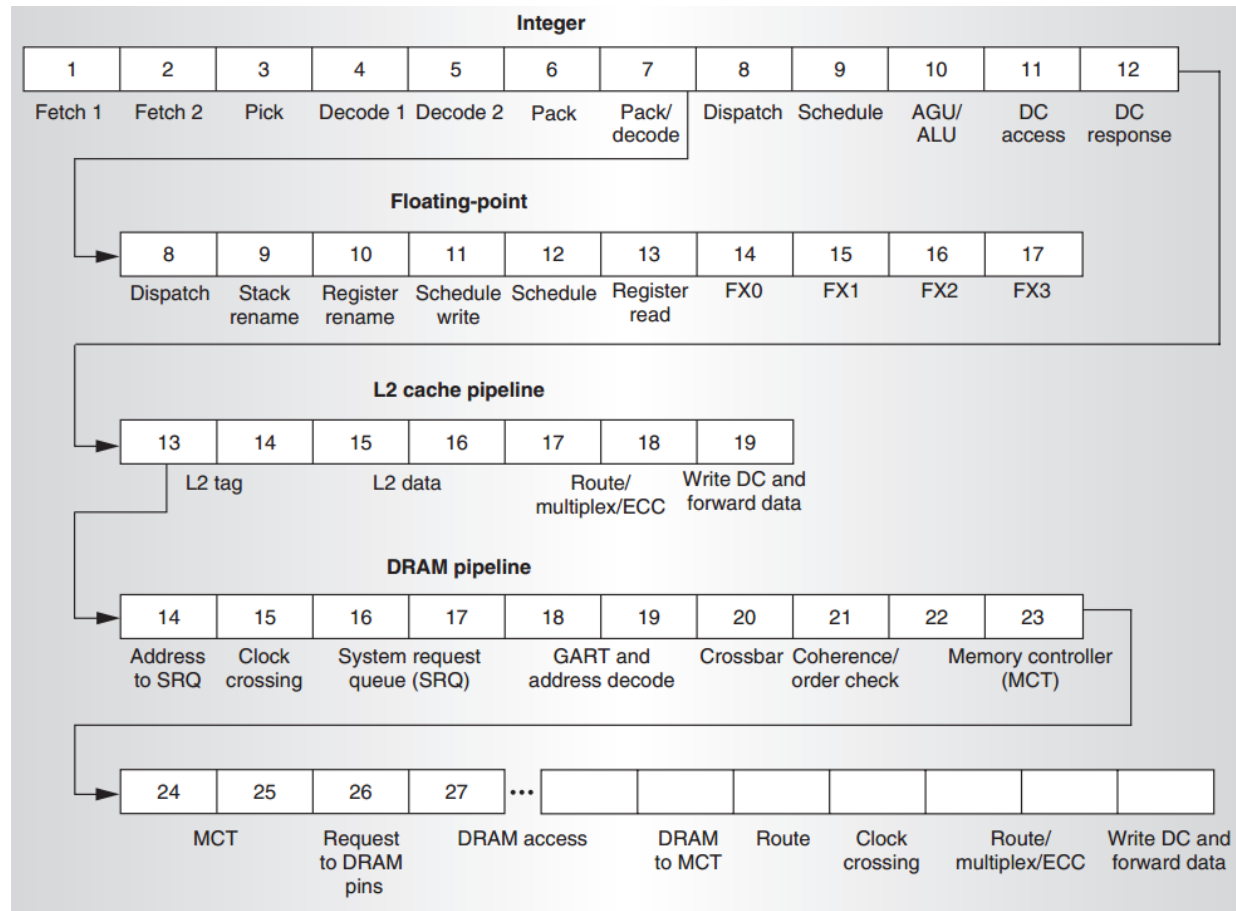
## “Opteron Family”

SledgeHammer(K8): 1 core

Denmark (K9): 2 cores

Barcelona (K10): 4+ cores

# AMD Opteron pipeline



Fetch & decode: 7 stages

Integer-ex pipeline: 12 stages (L1-Data cache access at 11 stage)

FP-ex pipeline: 17 stages

DRAM access: off-chip

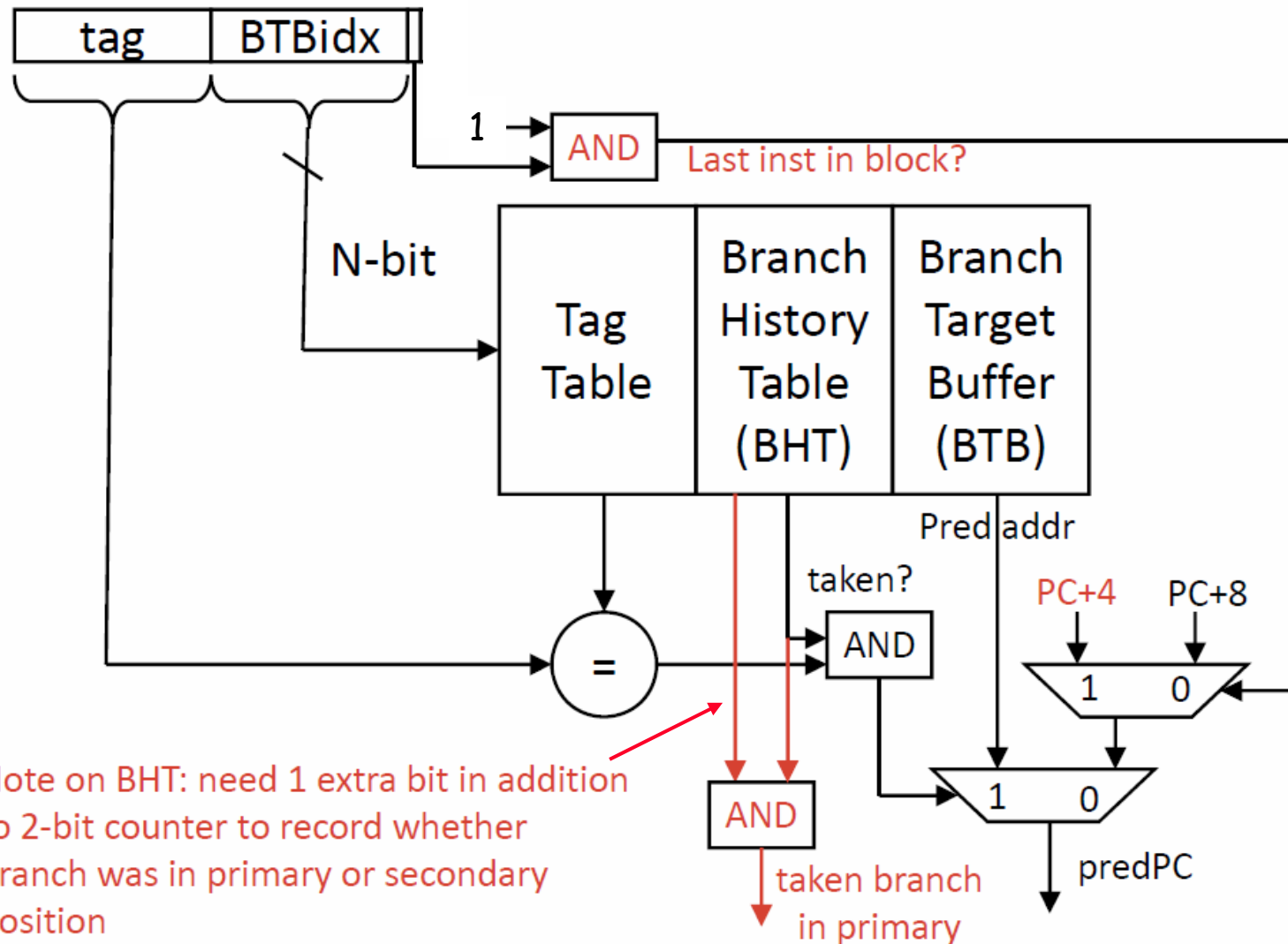


# Branch prediction in Superscalar CPU?

- ◆ “Superscalar” processors
  - Designed to execute more than 1 instruction-per-cycle
  - **Multiple** instruction fetch, decode, execution, completion per cycle  
: Wide pipeline or Multiple pipeline.
  - Modern CPUs are 4~8 way superscalar CPUs.
- ◆ Consider a “2-instruction” fetch scenario
  - (Case 1) Both insts are not-taken control-flow insts.
    - $\text{nextPC (nPC)} = \text{PC} + 8$
  - (Case 2) One of the insts is a taken control flow inst
    - $\text{nPC} = \text{predicted target address}$
  - (Case 3) Both instructions are control-flow insts.
    - Prediction based on the 1<sup>st</sup> branch’s “predicted” outcome.
    - If the 1<sup>st</sup> instruction is the predicted taken branch  
→ Must flush 2<sup>nd</sup> instruction which was “already fetched”



# Branch Predictor for 2-way Superscalar



# Question?

Announcements:

Reading: Read assigned papers!

Handouts: None