

Hardware Speculation: The Key to High Performance in Modern Processors

Shoaib Akram

shoaib.akram@anu.edu.au



Australian
National
University

Instruction-Level Parallelism (ILP)

Instruction-Level Parallelism (ILP)

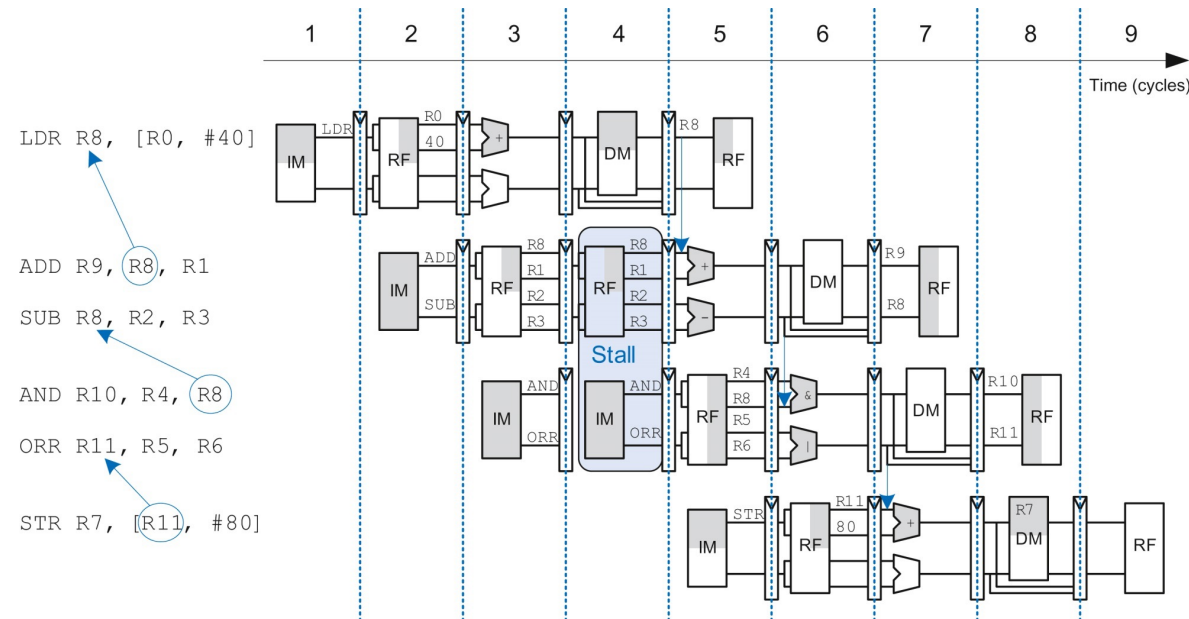
- Overlapping the execution of instructions is called instruction-level parallelism
- We have seen exploitation of ILP
 - In-order scalar pipeline (**partial overlap**)
 - In-order superscalar pipeline (**full overlap**)
- Full overlap requires multiple functional units (and datapaths)
- In general, ILP processors exploit the fact that many of the instructions in a sequential program do not depend on the instructions that immediately precede them

Limitation of In-Order Pipelines

- **In-order:** Instructions are fetched, decoded, and executed in program order
- CPU does no **reordering** of instructions to avoid stalls or execute independent instructions during a cache miss
- The simple **in-order** pipelines we have seen exploit **limited** ILP
- **They stall because of :**
 - Cache misses (100+ cycles)
 - Floating point multiply and divide (many cycles)
 - Load-use hazard
 - Some types of branches

Recall: Superscalar: Impact of Dependencies

- Example of program with data dependencies



- The program requires **5 cycles** to **issue six instructions** with an **IPC of 1.2**

More Serious Problem with an In-Order Pipeline

- Consider the following scenario (assume realistic memory latency)

ADD	R0,	R0,	#4
LDR	R1,	[R0,	#0]
ADD	R2,	R2,	#1
ADD	R3,	R3,	#2
ADD	R4,	R3,	#1
SUB	R5,	R1,	#1

- Load misses in the cache (100 cycles penalty to access memory)
- LDR must stall. (The dependent instruction on LDR is SUB)
- All subsequent instructions that do not depend on LDR also stall

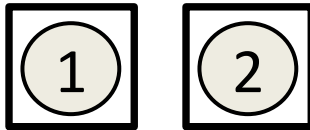
Dependences and Hazards

- To transition from in-order to out-of-order, we need to understand **two** new forms of data hazards
- **Recall:** Dependence is a program's property
- **Recall:** Hazard is a microarchitecture property
 - We have seen
 - Read-after write hazard (due to **true dependences**)
 - Control hazard (due to **branches** in programs)

Example Sequence 1: No dependences

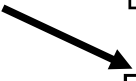
1. LDR R2, [R6, #0]
2. ADD R3, R4, R5

Two independent instructions



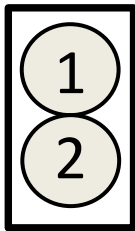
Example Sequence 1: True dependence

1. LDR R2, [R6, #0]
2. ADD R3, R2, R5



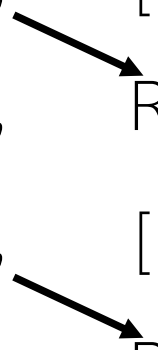
Data or true dependence
i2 needs the result of i1

A single (dependent) instruction chain



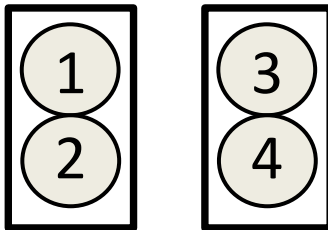
Example Sequence 3: True dependencies

1. LDR R2, [R6, #0]
2. ADD R3, R2, R5
3. LDR R4, [R6, #0]
4. ADD R7, R4, R9



Data or true dependence
i2 needs the result of **i1**
i4 needs the result of **i3**

Two independent instruction chains




New Types of Hazards

- The next two hazards are relevant to a processor that can execute instructions whenever operands are ready
 - I.e., an out-of-order processor
- We don't know how out-of-order execution can be done yet

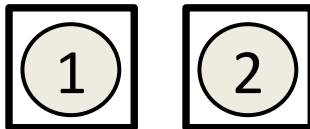
Example Sequence 4: Anti dependence

1. LDR R2, [R6, #0]
2. ADD R6, R3, R5



False (anti) dependence
i2 needs to write what
i1 needs to read

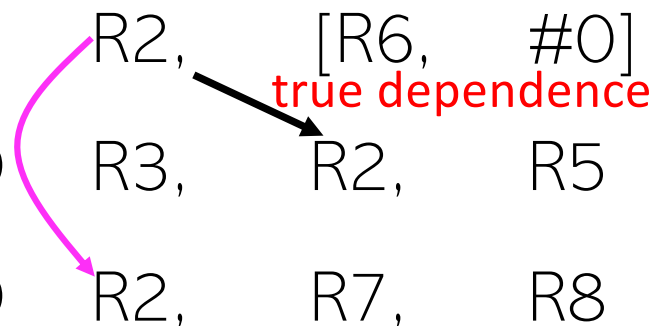
Two independent instructions



- Limitation of number of registers in the ISA, but **ILP** exists
- We can rename register R6 in **the second instruction** and **eliminate** the (false) dependency

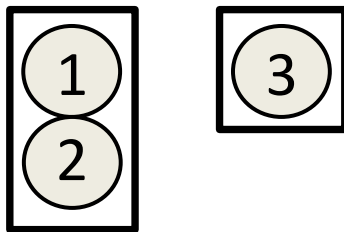
Example Sequence 5: Output dependence

1. LDR R2, [R6, #0]
2. ADD R3, R2, R5
3. ADD R2, R7, R8



Output dependence
i1 and i3 wants to write
to the same register

Instruction chain (i1→i2) + independent instruction (i3)



- Limitation of # registers, but ILP exists
- We can rename register R2 in **(3)** and execute (3) earlier than 1 – 2 chain

Dependences and Hazards

- True dependence results in:
 - Read-after-write hazard (RAW)
- Anti-dependence results in:
 - Write-after-read hazard (WAR)
- Output dependence results in:
 - Write-after-write hazard (WAW)
- **Single-cycle CPU:** Each instruction takes one cycle; one instruction at any time
 - None of the dependences result in a hazard
- **In-order pipeline:** Multiple instructions in different stages (possibility of RAW)
- **Out-of-order:** ALL BETS ARE OFF!

What we have learnt so far

- Dependences exist in real programs due to
 - Programmer's intention
 - Limited number of registers
 - Branches
- Dependences lead to hazards (microarchitecture dependent)
- True dependences create a chain of instructions that cannot be reordered
- False dependences can be eliminated to uncover greater ILP

Instruction Scheduling

- A straight-line instruction sequence can be executed in different orderings
 - Each ordering is an instruction schedule
 - Must respect true dependences

ADD	R0,	R0,	#4
LDR	R1,	[R0,	#0]
ADD	R2,	R2,	#1
ADD	R3,	R3,	#2
ADD	R4,	R3,	#1
SUB	R5,	R1,	#1

- Original sequence

ADD	R0,	R0,	#4
LDR	R1,	[R0,	#0]
SUB	R5,	R1,	#1
ADD	R2,	R2,	#1
ADD	R3,	R3,	#2
ADD	R4,	R3,	#1

- Valid reordering

LDR	R1,	[R0,	#0]
ADD	R0,	R0,	#4
ADD	R2,	R2,	#1
ADD	R3,	R3,	#2
ADD	R4,	R3,	#1
SUB	R5,	R1,	#1

- Invalid reordering

Instruction Scheduling

- Some reorderings or schedules exhibit high ILP than others depending on the processor
- Consider the following two instruction schedules

LDR	R0,	[R7, #4]
ADD	R1,	R0, #0
LDR	R2,	[R7, #8]
ADD	R3,	R2, #2

LDR	R0,	[R7, #4]
LDR	R2,	[R7, #8]
ADD	R1,	R0, #0
ADD	R3,	R2, #2

- **On a processor that**
 - stalls due to a load-use hazard,
 - each memory access takes 100 cycles,
 - and multiple memory accesses can be resolved concurrently
 - Which schedule is better?
 - What about a single-cycle CPU?

A Historical Debate

- Historical (**ongoing**) debate: How best to exploit ILP?
- **Dynamically in hardware** (dynamic = during program execution)
 - Portable: no need to recompile code to run on a different processor
 - Hardware has more knowledge of program, e.g., loop counters, branch directions, program inputs
 - **Power, energy, and security issues**
- **Statically in software** (static = during program compilation)
 - Compiler can do whole-program optimizations
 - Compiler has more time to analyze code to find ILP
 - **Compiler does not know about program inputs, so it needs to guess too much**
 - **A commercial failure**

ILP Exploitation: A Taxonomy

- **Statically scheduled superscalar processor**
 - Compiler schedules instructions during program creation
 - Hardware does no **reordering of instructions**
 - Dependency checks in hardware
 - Compiler can rely on hardware for correct execution
- **VLIW (Very Long Instruction Word) processor**
 - Static scheduling by compiler. Instruction words are very large. Up to 28 insts. in a bundle
 - Compiler does “**very smart and deep**” program analysis to construct “good instruction schedules with high ILP”
 - Conceptually same as above. “Smart compiler, dumb hardware.”
 - **No dependency checking in hardware**
- **Dynamically scheduled superscalar processor**
 - Hardware does scheduling during program execution (can reorder instructions for ILP)
 - **Hardware can construct different “instruction schedules” based on different executions of the same set of basic blocks (different branch outcomes)**

VLIW Architectures (Very Long Instruction Word)

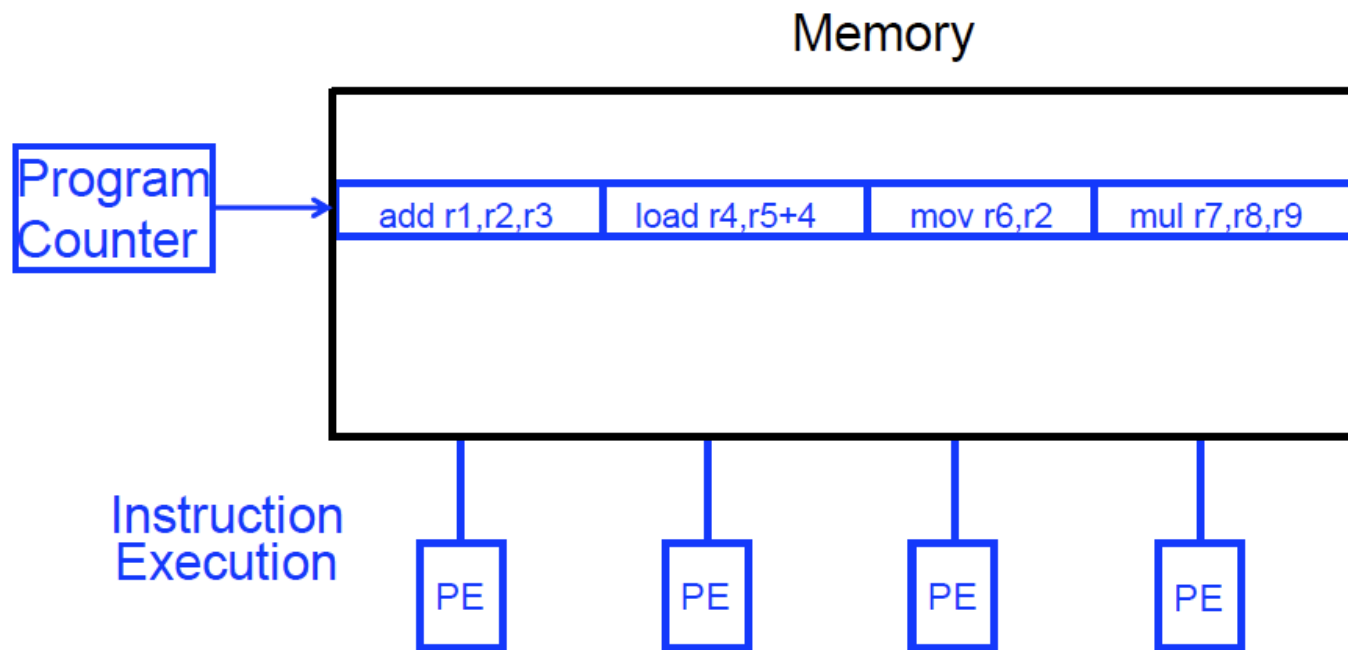
VLIW Concept

- Superscalar
 - **Hardware** fetches multiple instructions and checks dependencies between them

- VLIW (Very Long Instruction Word)
 - **Software (compiler) packs independent instructions** in a larger “instruction bundle” to be fetched and executed concurrently
 - Hardware fetches and executes the instructions in the bundle concurrently

- **No need for hardware dependency checking** between concurrently-fetched instructions **in the VLIW model**
 - **Simple hardware, complex compiler**

VLIW Concept



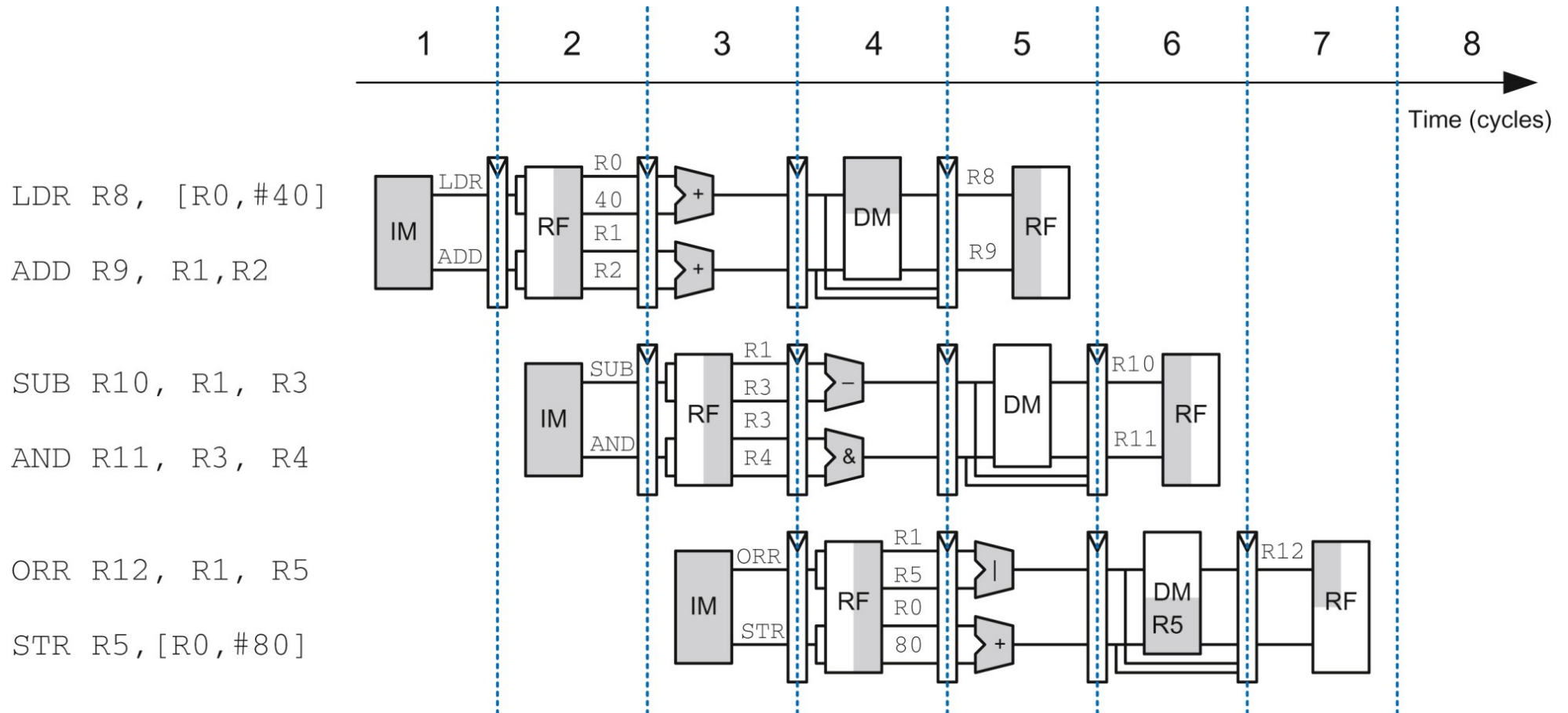
- Fisher, “**Very Long Instruction Word architectures and the ELI-512,**” ISCA 1983.
 - ELI: Enormously longword instructions (512 bits)

VLIW (Very Long Instruction Word)

- A very long instruction word consists of **multiple independent instructions packed together by the compiler**
 - Packed instructions can be logically unrelated (contrast with SIMD/vector processors)
- Idea: Compiler finds independent instructions and statically schedules (i.e. packs/bundles) them into a single VLIW instruction
- Traditional VLIW Characteristics
 1. Multiple instruction fetch/execute, multiple functional units
 2. All instructions in a bundle are executed in lock step
 3. Instructions in a bundle statically aligned to be directly supplied into the functional units

VLIW Example: 2-Wide Bundles

- Example program where IPC = 2 is possible



- We saw this example before (recall superscalar)

VLIW Lock-Step Execution

- Lock-step (all or none) execution
 - ❑ If any operation in a VLIW instruction stalls, all concurrent operations stall

- In a **truly VLIW machine**:
 - ❑ the compiler handles all dependency-related stalls
 - ❑ hardware does **not** perform dependency checking
 - ❑ What about variable latency operations? Memory stalls?

VLIW Philosophy & Principles (Self-Study)

- Philosophy similar to RISC (simple instructions and hardware)
 - Except “multiple instructions in parallel: in VLIW

- RISC (John Cocke+, 1970s, IBM 801 minicomputer)
 - Compiler does the hard work to translate high-level language code to simple instructions (John Cocke: control signals)
 - And, to reorder simple instructions for high performance
 - Hardware does little translation/decoding → very simple

- VLIW (Josh Fisher, ISCA 1983)
 - Compiler does the hard work to find instruction level parallelism
 - Hardware stays as simple as possible
 - Executes each instruction in a bundle in lock step
 - Simple → higher frequency, easier to design, low power

Our Focus

- **Dynamically scheduled superscalar processor**
 - Hardware does “dynamic scheduling” during program execution
 - Can reorder instructions to extract maximum ILP
 - Hardware can construct different “instruction schedules” based on different executions of the same sequence of instructions
 - To account for change in branch behavior
- Dynamically scheduled processors extract ILP by gathering instructions in a large instruction window and then performing dataflow analysis
 - If operands ready and no hazards, execute the instruction
 - Multiple independent instruction chains are in execution in any cycle

From In-Order to Out of Order

Problem with In-Order Pipeline

- Consider the following scenario

ADD	R0,	R0,	#4
LDR	R1,	[R0,	#0]
ADD	R2,	R2,	#1
ADD	R3,	R3,	#2
ADD	R4,	R3,	#1
SUB	R5,	R1,	#1

- Two options to exploit more ILP:**
 - Stall until dependent instruction SUB is encountered
(aggressive but still in order)
 - Keep executing independent instructions (ones that come after SUB) as long as their operands are ready (out of order)

Two Types of In-Order Pipelines

- **Stall-on-miss** (**simple issue policy**)
 - Stall the pipeline on a **long-latency event** such as a cache miss
 - **Too naive. Extremely low-power environments.**
 - Let's make it better
 - Even the ARM 5-stage pipeline we saw previously was better than this
- **Stall-on-use** (**aggressive issue policy**)
 - Stall the pipeline on a **RAW** hazard (when the **“use”** instruction is encountered)
 - Need extra logic to track pending operation and its dependents
 - Many functional units yet only one new instruction can begin execution in any cycle (more on this later)

Stall-on-Use Pipeline

- We will use the aggressive in-order pipeline to understand the problem with in-order pipelines
- Instruction **issue policy** (advancement from RF read to execute)
 - If RAW hazard, then stall
 - If WAW hazard, then stall
 - WAR is not possible in in-order pipeline

Stall-on-Use Pipeline

- Assumptions about **execute** stage
 - **Non-blocking execute stage (multiple functional units)**
 - **Pipelined** functional units (concurrent instruction execution)
 - Load is divide into:
 - address generation (**agen**)
 - data cache access (**D\$**)
 - memory access (**if load miss or cache miss**)
- This slide is the key distinction between ARM 5-stage pipeline we saw before
 - One instruction is issued **in order** every cycle to the functional units, but many of them can be in execution in any cycle (**limiting the structural dependency**)

Stall-on-Use Pipeline

- Decode and register read stages are separate
- Register read stage also implements the issue policy

Fetch

Decode

Register Read

Execute

+	Big	agen
Tiny ALU	ALU	D\$
		Mem

Writeback

Fetch

Decode

Register Read

Execute

+	Big	agen
Tiny	ALU	D\$
ALU		Mem

Writeback

Assumptions

Scalar:

- *fetch 1 inst/cycle*
- *decode 1 inst/cycle*
- *issue 1 inst/cycle to a function unit*



Assumptions

Issue logic:

- **RAW hazard:** Instruction stalls if its source registers are not ready
- **WAW:** Instruction stalls if its destination register is “busy”
- **WAR hazard:** Not a problem in in-order pipelines. In-order issue ensures read by first instruction happens before write by second instruction

Scenario 1: load miss followed by independent instructions



i1

Scenario 1: load miss followed by independent instructions

```
i1: LDR    R2, [R1, #0]
```

```
i2: ADD    R4, [R3, #1]
```

```
i3: ADD    R6,  R5,  #2
```

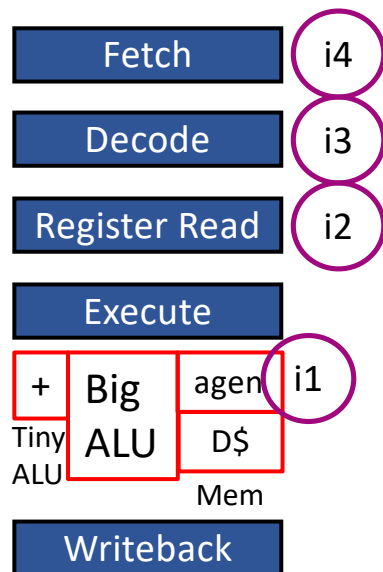
i4: ADD R7, R6, #3

[illegible]



Scenario 1: load miss followed by independent instructions

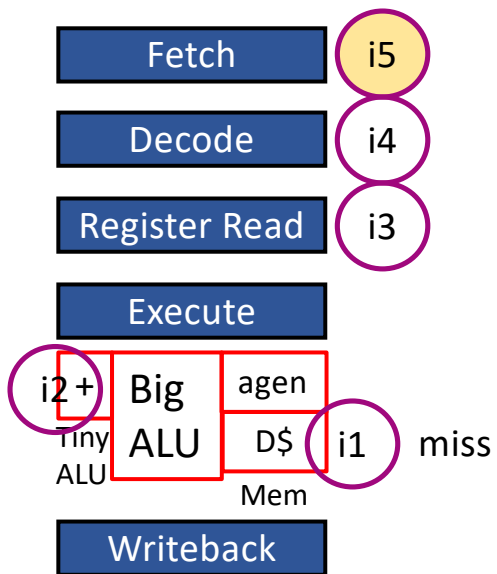
[illegible]



i4

Scenario 1: load miss followed by independent instructions

[illegible]



Scenario 1: **load miss** followed by **independent** instructions

i1: LDR R2, [R1, #0]

i2: ADD R4, [R3, #1]

i3: ADD R6, R5, #2

i4: ADD R7, R6, #3

	1	2	3	4	5	6	7	8	9	10	11	12	13
i1	FE	DE	RR	EX@	EX _{D\$}	...miss...							
i2		FE	DE	RR	EX								
i3			FE	DE	RR								
i4				FE	DE								



Scenario 1: load miss followed by independent instructions

i1: LDR R2, [R1, #0]

i2: ADD R4, [R3, #1]

i3: ADD R6, R5, #2

i4: ADD R7, R6, #3

	1	2	3	4	5	6	7	8	9	10	11	12	13
i1	FE	DE	RR	EX@	EXD\$...miss...							
i2		FE	DE	RR	EX	WB							
i3			FE	DE	RR	EX							
i4				FE	DE	RR							



Scenario 1: **load miss** followed by **independent** instructions

i1: LDR R2, [R1, #0]

i2: ADD R4, [R3, #1]

i3: ADD R6, R5, #2

i4: ADD R7, R6, #3

	1	2	3	4	5	6	7	8	9	10	11	12	13
i1	FE	DE	RR	EX _@	EX _{D\$}	...miss...							
i2		FE	DE	RR	EX	WB							
i3			FE	DE	RR	EX	WB						
i4				FE	DE	RR	EX						



Scenario 1: load miss followed by independent instructions

i1: LDR R2, [R1, #0]

i2: ADD R4, [R3, #1]

i3: ADD R6, R5, #2

i4: ADD R7, R6, #3

	1	2	3	4	5	6	7	8	9	10	11	12	13
i1	FE	DE	RR	EX _@	EX _{D\$}	...miss...							
i2		FE	DE	RR	EX	WB							
i3			FE	DE	RR	EX	WB						
i4				FE	DE	RR	EX	WB					

Fetch

Decode

Register Read

Execute

+ Big agen
Tiny ALU D\$
ALU Mem

Writeback

i1

Scenario 1: **load miss** followed by **independent** instructions

i1: LDR R2, [R1, #0]

i2: ADD R4, [R3, #1]

i3: ADD R6, R5, #2

i4: ADD R7, R6, #3

	1	2	3	4	5	6	7	8	9	10	11	12	13
i1	FE	DE	RR	EX _@	EX _{D\$}	...miss...				WB			
i2		FE	DE	RR	EX	WB							
i3			FE	DE	RR	EX	WB						
i4				FE	DE	RR	EX	WB					

Fetch

Decode

Register Read

Execute

+ Big agen
Tiny ALU D\$
ALU Mem

Writeback

i1

Scenario 1: **load miss** followed by **independent** instructions

i1: LDR R2, [R1, #0]

i2: ADD R4, [R3, #1]

i3: ADD R6, R5, #2

i4: ADD R7, R6, #3

	1	2	3	4	5	6	7	8	9	10	11	12	13
i1	FE	DE	RR	EX@	EX _{D\$}	...miss...				WB			
i2		FE	DE	RR	EX	WB							
i3			FE	DE	RR	EX	WB						
i4				FE	DE	RR	EX	WB					

Aggressive in-order pipeline does work “underneath a load miss” to **hide** the memory latency

Scenario 2: Load miss followed by dependent instruction,
followed by independent instructions



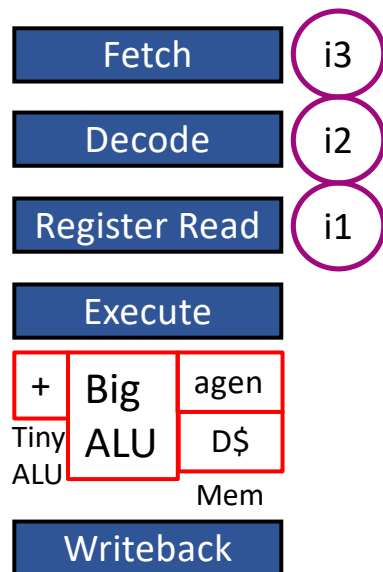
i1

[illegible]



Scenario 2: load miss followed by dependent instruction, followed by independent instructions

[illegible]



i3

Scenario 2: load miss followed by dependent instruction, followed by independent instructions

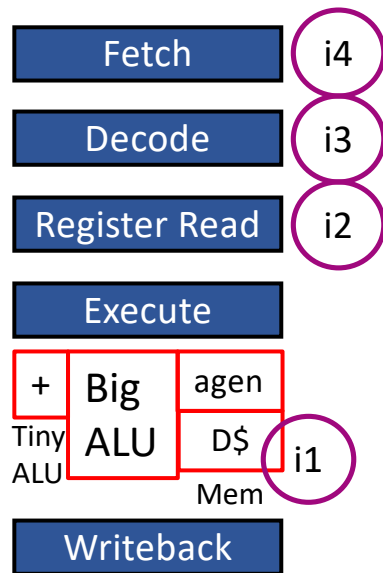
[illegible]



i4

Scenario 2: load miss followed by dependent instruction, followed by independent instructions

[illegible]

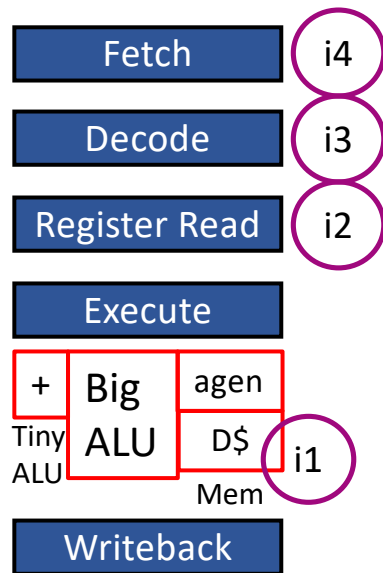


Scenario 2: load miss followed by dependent instruction, followed by independent instructions

i1: LDR R2, [R1, #0]
 i2: ADD R4, R2, #1
 i3: ADD R6, R5, #2
 i4: ADD R7, R6, #3

Arrows indicate data dependencies: from i1 to i2 (R2) and from i3 to i4 (R6).

	1	2	3	4	5	6	7	8	9	10	11	12	13
i1	FE	DE	RR	EX _@	EX _{D\$}	...miss...							
i2		FE	DE	RR	RR								
i3			FE	DE	DE								
i4				FE	FE								

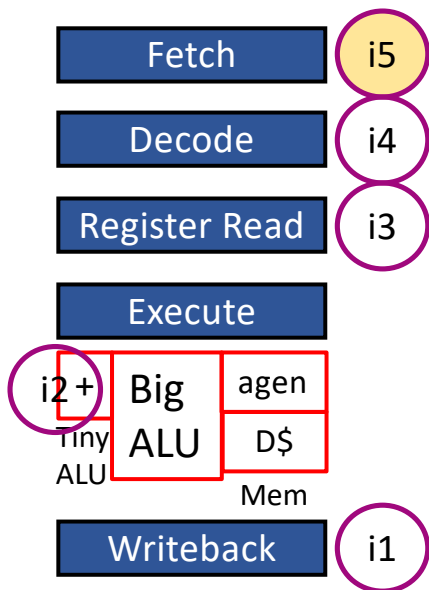


Scenario 2: load miss followed by dependent instruction, followed by independent instructions

i1: LDR R2, [R1, #0]
 i2: ADD R4, R2, #1
 i3: ADD R6, R5, #2
 i4: ADD R7, R6, #3

(Blue arrows indicate data dependencies from R2 in i2 to R2 in i1, and from R6 in i4 to R6 in i3.)

	1	2	3	4	5	6	7	8	9	10	11	12	13
i1	FE	DE	RR	EX _@	EX _{D\$}	...miss...							
i2		FE	DE	RR	RR	RR	RR	RR	RR				
i3			FE	DE	DE	DE	DE	DE	DE				
i4				FE	FE	FE	FE	FE	FE				



Scenario 2: load miss followed by dependent instruction, followed by independent instructions

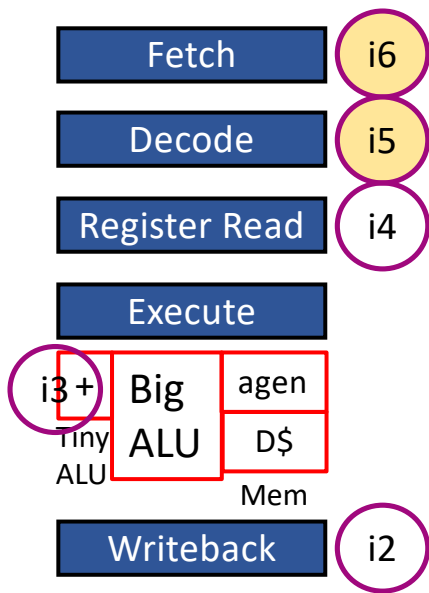
i1: LDR R2, [R1, #0]

i2: ADD R4, R2, #1

i3: ADD R6, R5, #2

i4: ADD R7, R6, #3

	1	2	3	4	5	6	7	8	9	10	11	12	13
i1	FE	DE	RR	EX _@	EX _{D\$}	...miss...				WB			
i2		FE	DE	RR	RR	RR	RR	RR	RR	EX			
i3			FE	DE	DE	DE	DE	DE	DE	RR			
i4				FE	FE	FE	FE	FE	FE	DE			

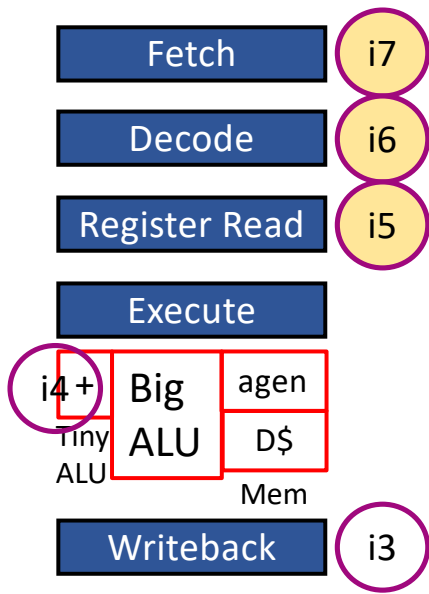


Scenario 2: load miss followed by dependent instruction, followed by independent instructions

i1: LDR R2, [R1, #0]
 i2: ADD R4, R2, #1
 i3: ADD R6, R5, #2
 i4: ADD R7, R6, #3

Blue arrows indicate data dependencies: from R2 in i1 to R2 in i2, and from R6 in i3 to R6 in i4.

	1	2	3	4	5	6	7	8	9	10	11	12	13
i1	FE	DE	RR	EX _@	EX _{D\$}	...miss...				WB			
i2		FE	DE	RR	RR	RR	RR	RR	RR	EX	WB		
i3			FE	DE	DE	DE	DE	DE	DE	RR	EX		
i4				FE	FE	FE	FE	FE	FE	DE	RR		

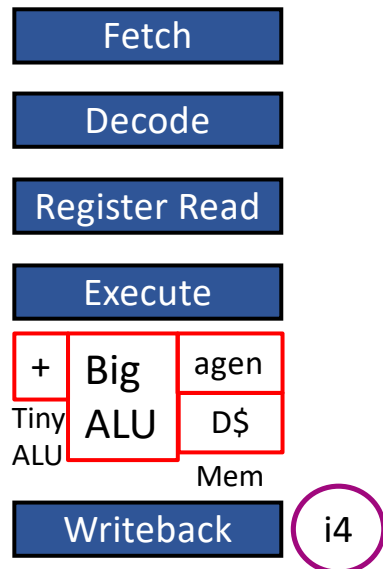


Scenario 2: load miss followed by dependent instruction, followed by independent instructions

i1: LDR R2, [R1, #0]
 i2: ADD R4, R2, #1
 i3: ADD R6, R5, #2
 i4: ADD R7, R6, #3

Arrows indicate data dependencies: from R2 in i1 to R2 in i2, and from R6 in i3 to R6 in i4.

	1	2	3	4	5	6	7	8	9	10	11	12	13
i1	FE	DE	RR	EX _@	EX _{D\$}	...miss...				WB			
i2		FE	DE	RR	RR	RR	RR	RR	RR	EX	WB		
i3			FE	DE	DE	DE	DE	DE	DE	RR	EX	WB	
i4				FE	FE	FE	FE	FE	FE	DE	RR	EX	



Scenario 2: load miss followed by dependent instruction, followed by independent instructions

i1: LDR R2, [R1, #0]

i2: ADD R4, R2, #1

i3: ADD R6, R5, #2

i4: ADD R7, R6, #3

	1	2	3	4	5	6	7	8	9	10	11	12	13
i1	FE	DE	RR	EX _@	EX _{D\$}	...miss...				WB			
i2		FE	DE	RR	RR	RR	RR	RR	RR	EX	WB		
i3			FE	DE	DE	DE	DE	DE	DE	RR	EX	WB	
i4				FE	FE	FE	FE	FE	FE	DE	RR	EX	WB



Scenario 2: load miss followed by dependent instruction, followed by independent instructions

i1: LDR R2, [R1, #0]

i2: ADD R4, R2, #1

i3: ADD R6, R5, #2

i4: ADD R7, R6, #3

	1	2	3	4	5	6	7	8	9	10	11	12	13
i1	FE	DE	RR	EX _@	EX _{D\$}	...miss...				WB			
i2		FE	DE	RR	RR	RR	RR	RR	RR	EX	WB		
i3			FE	DE	DE	DE	DE	DE	DE	RR	EX	WB	
i4				FE	FE	FE	FE	FE	FE	DE	RR	EX	WB

Even aggressive in-order pipeline cannot hide the latency of a load miss when confronted with a RAW hazard. Independent instructions wait needlessly, hindering ILP exploitation

What have we established?

- **In-order issue policy**
 - If a younger instruction has a **RAW** hazard with an older instruction (**must stall and it's ok!**)
 - What about instructions after it?
 - Some of the younger instructions may be independent
 - This is where the problem lies!

In-Order Issue Bottleneck

- i_2 must wait for i_1
 - i_2 depends on i_1 (chain of dependent instructions)
- i_3, i_4 need not wait for the i_1-i_2 chain
 - They are independent
- But the i_3-i_4 chain stalls
 - Key insight: *In-order issue translates into a structural hazard*
 - *RR stage (issue stage) blocked by the stalled i_2*
- *OOO pipeline unblocks RR (issue) using a new instruction buffer for stalled data-dependent instructions*
 - *A structure with many names: “Reservation stations”, “issue buffer”, “issue queue”, “scheduler”, “scheduling window”*

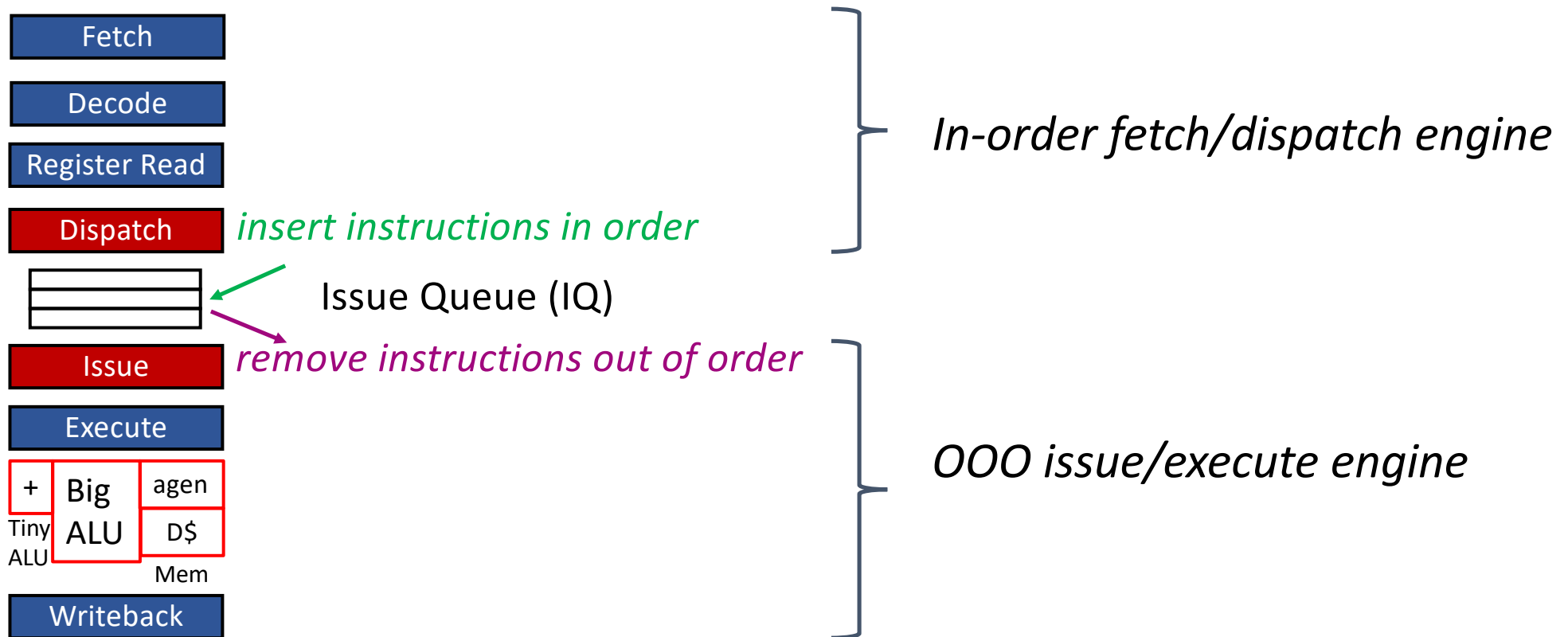
Nature of OOO

- **Out of order pipeline**
 - An instruction stalls if it has a **RAW hazard** with a previous instruction (that's ok)
 - Independent instructions after it do not stall: **they may issue out of program order**
- **Two alternatives for handling WAR and WAW**
 - Stall the pipeline (in-order-style, v.1)
 - Register renaming (optional optimization, v.2)

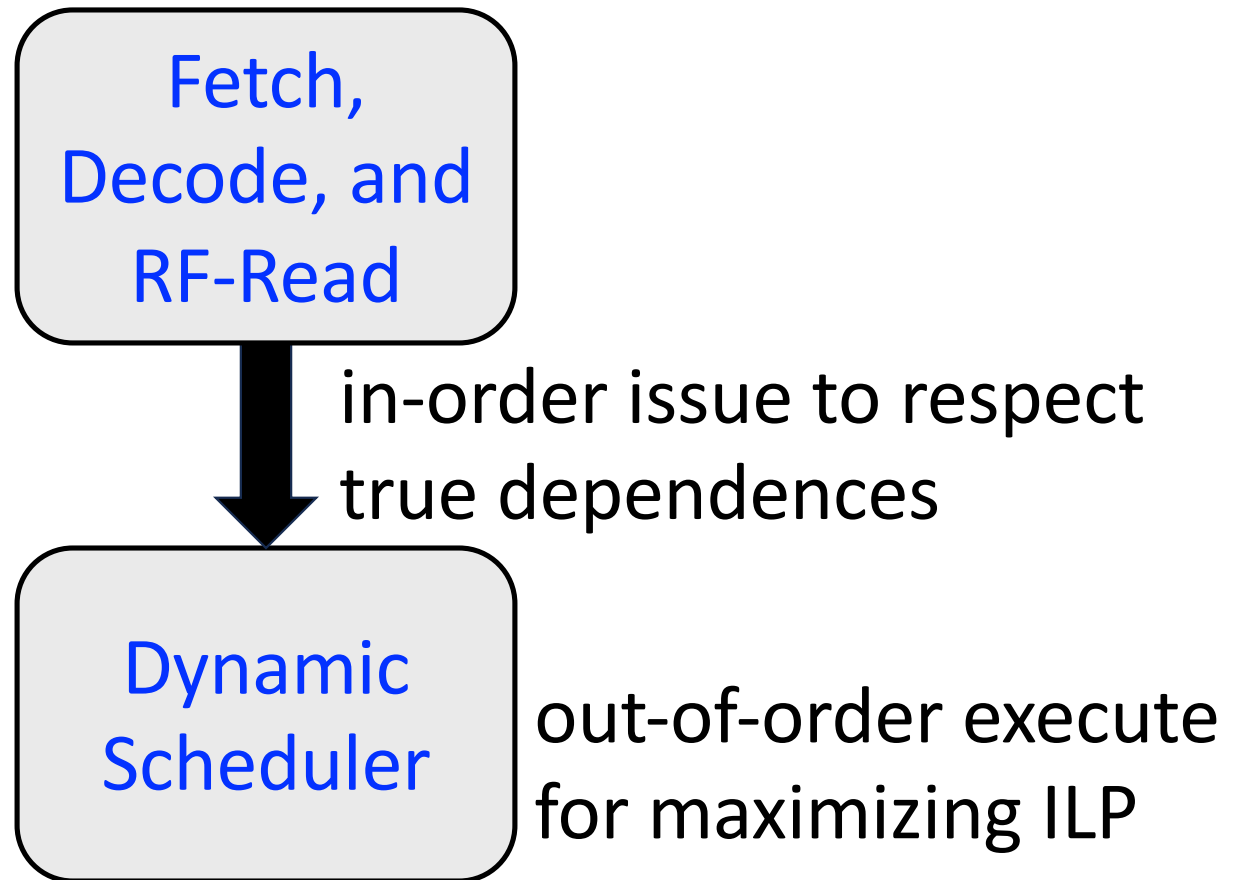
Issue Queue

- Stalled instructions do not impede instruction fetch
- Younger **ready** instructions issue and execute out of order with respect to older non-ready instructions
- Issue queue opens up the pipeline to future independent instructions
 - Tolerate long latencies (cache misses, floating point)
 - Exploit ILP (critical for superscalar)

Out-of-Order Scalar Pipeline (v.1)

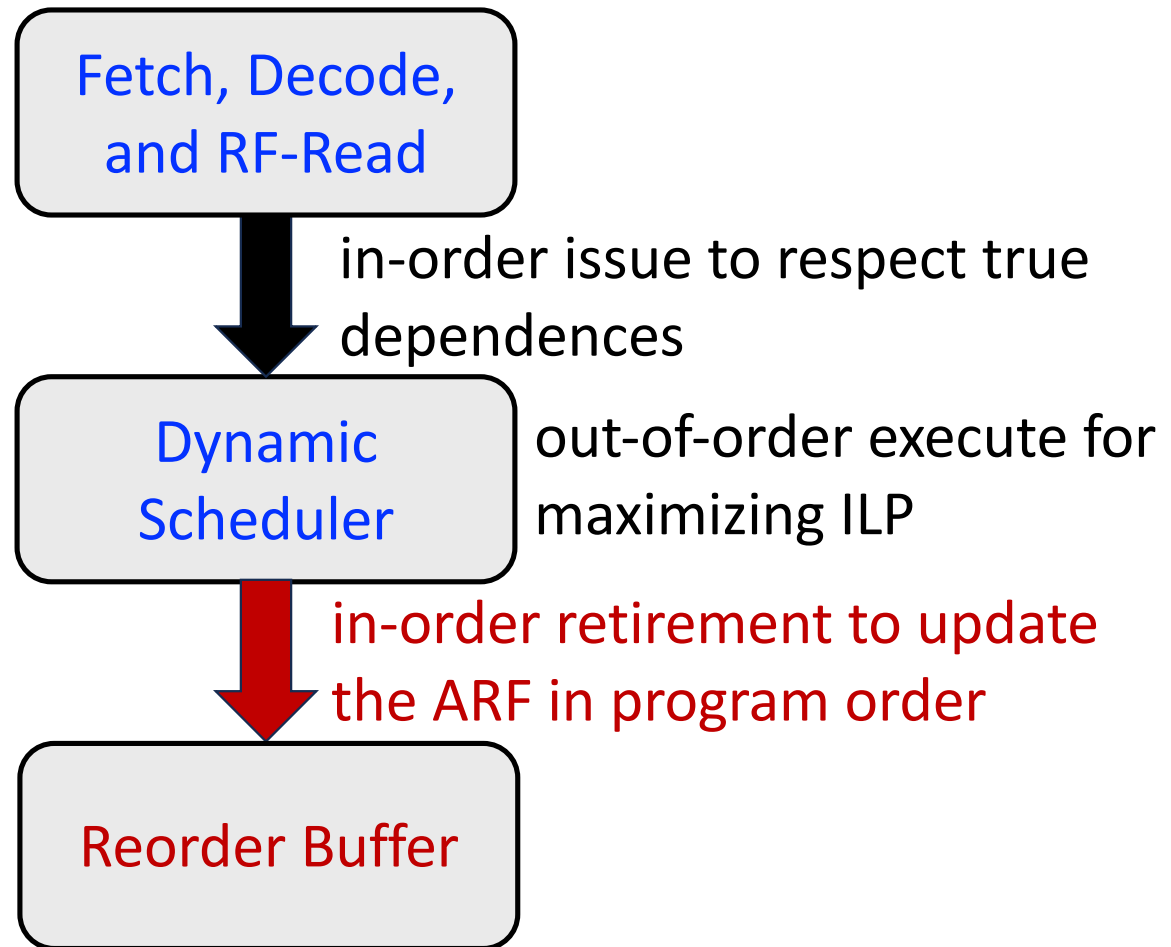


Out-of-Order Scalar Pipeline (v.1)



Out-of-Order Scalar Pipeline (v.2)

- We will look at this version later, first v.1



Dynamic Scheduling

Issue Queue

- Issue queue enables dynamically scheduled processors
 - **Dynamic scheduling:** Deciding which instructions to execute next , possibly reordering them to avoid stalls.
 - In a dynamically scheduled pipeline, instructions are issued **in-order** but can **bypass** each other and execute **out of order**

Real OOO Processors

- Many ALUs and functional units to maximize ILP exploitation
- Superscalar
 - 8-issue wide, for example
 - Meaning 8 instructions fetched, decoded, executed, every cycle (if ILP is available)
- Dynamically scheduled processors are superscalar, but we will assume scalar execution to simplify illustration of principles

Two Humps in a Modern Pipeline

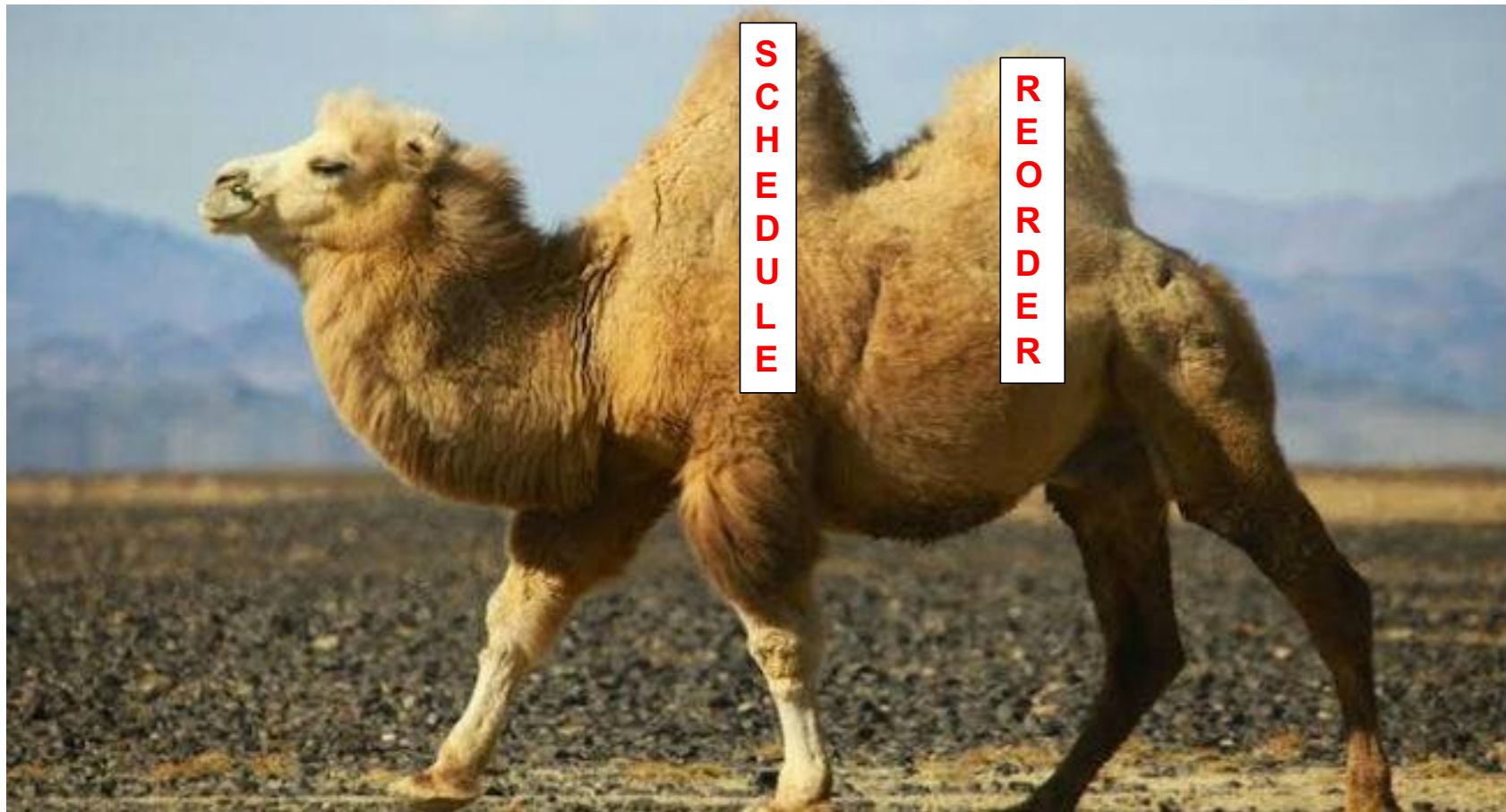
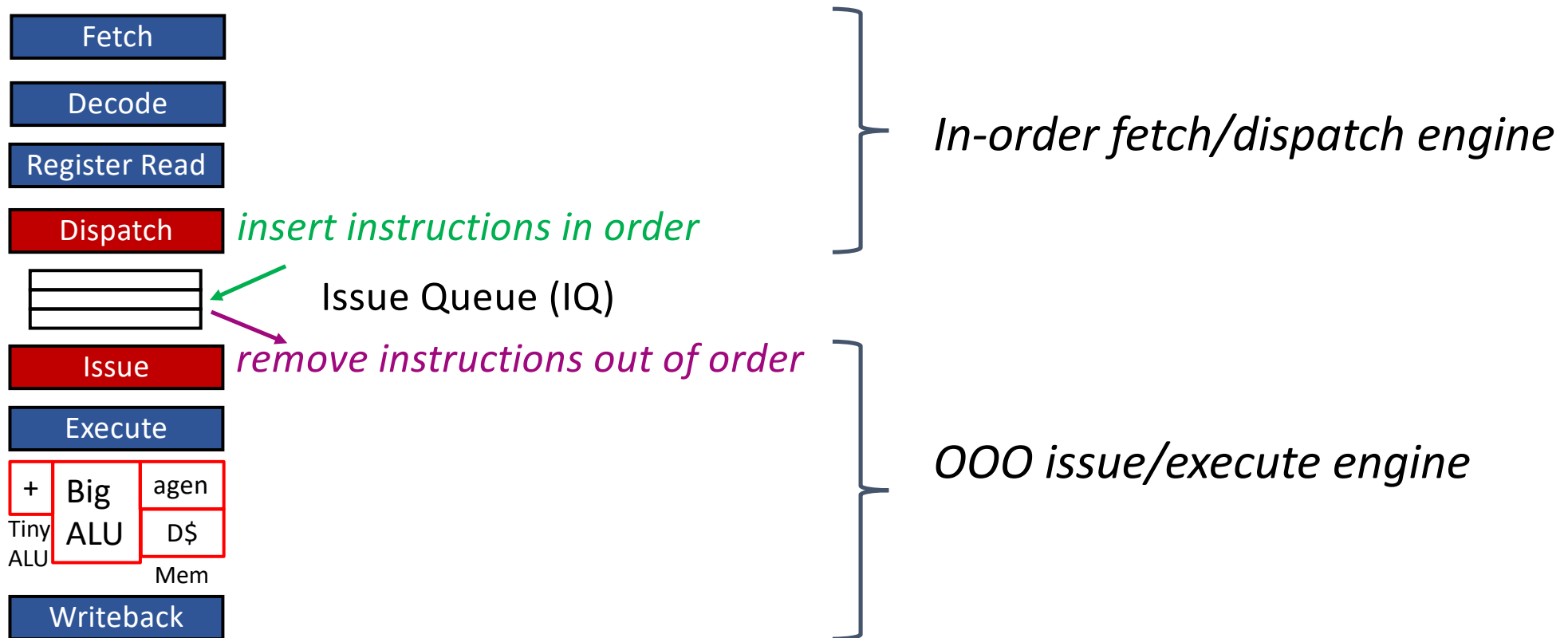
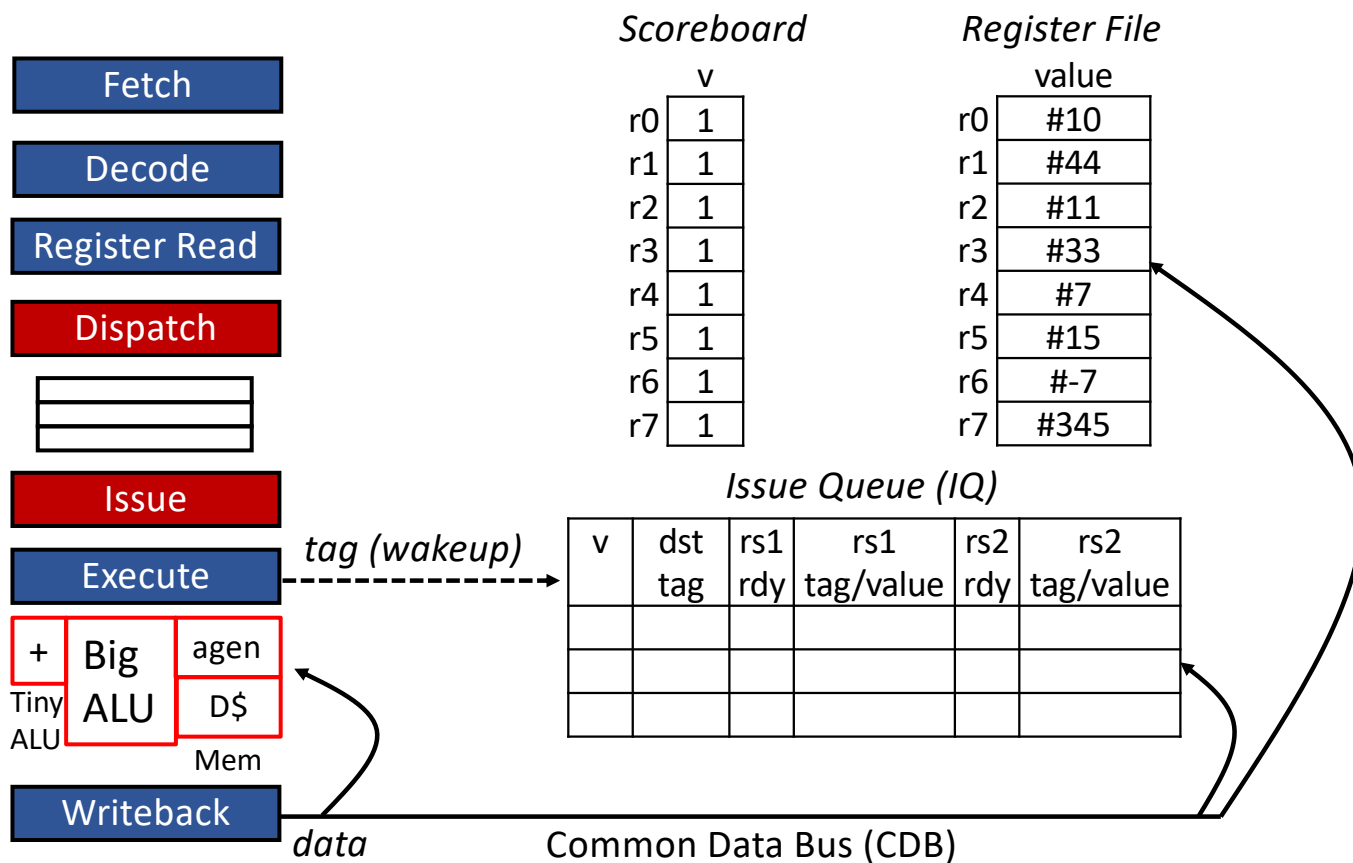


Photo credit: <http://true-wildlife.blogspot.ch/2010/10/bactrian-camel.html>

Out-of-Order Scalar Pipeline (v.1)





- The idea of a scoreboard was introduced by Control Data Corporation (CDC) in CDC6600 machine
- We will study next

CDC 6600 Scoreboard (Key Additions)

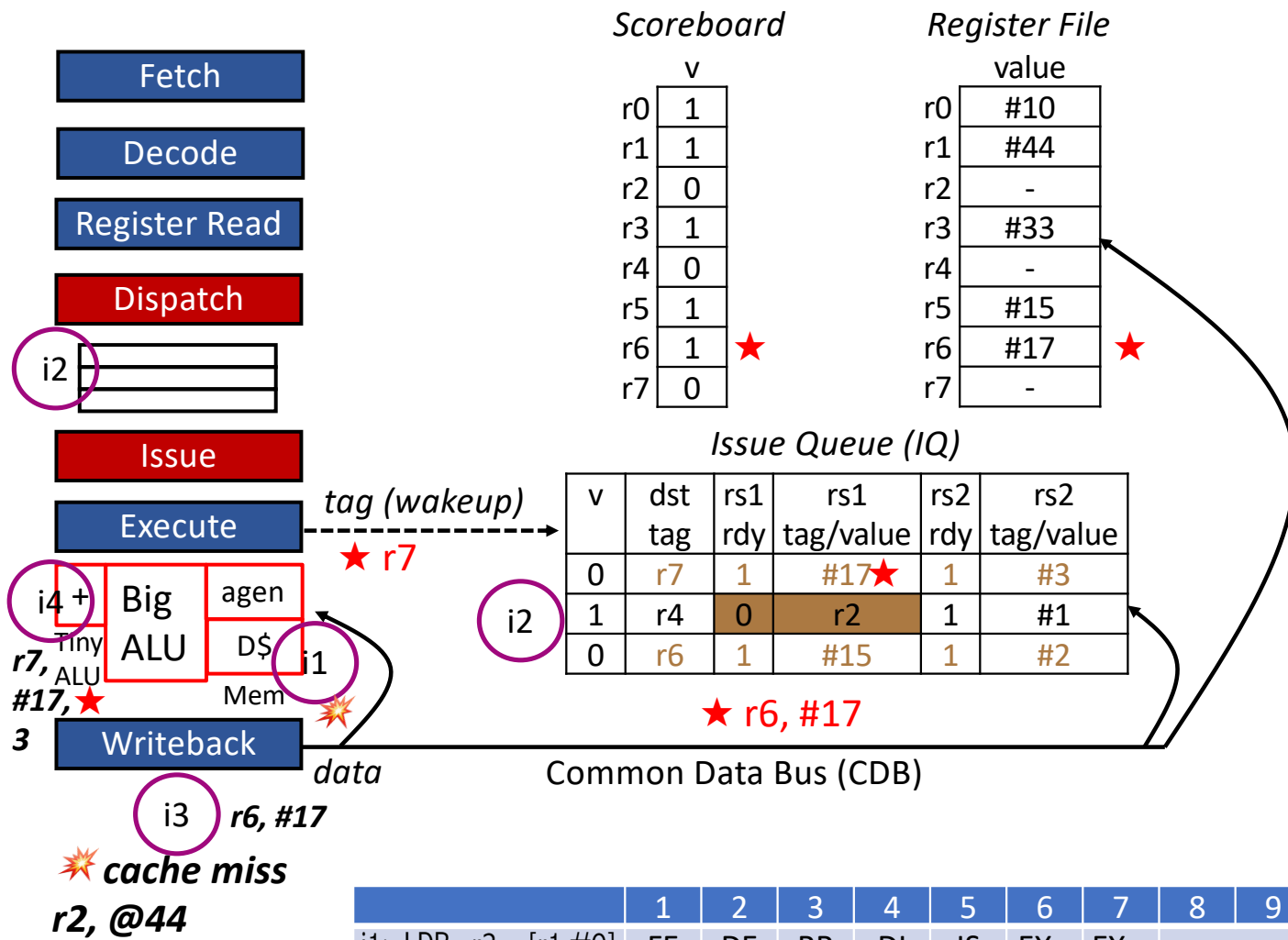
- Dispath stage
- Issue stage
- Scoreboard
- Common Data Bus (CDB)
- Instruction wakeup and select logic (issue stage)

CDC 6600 Scoreboard (Key Additions)

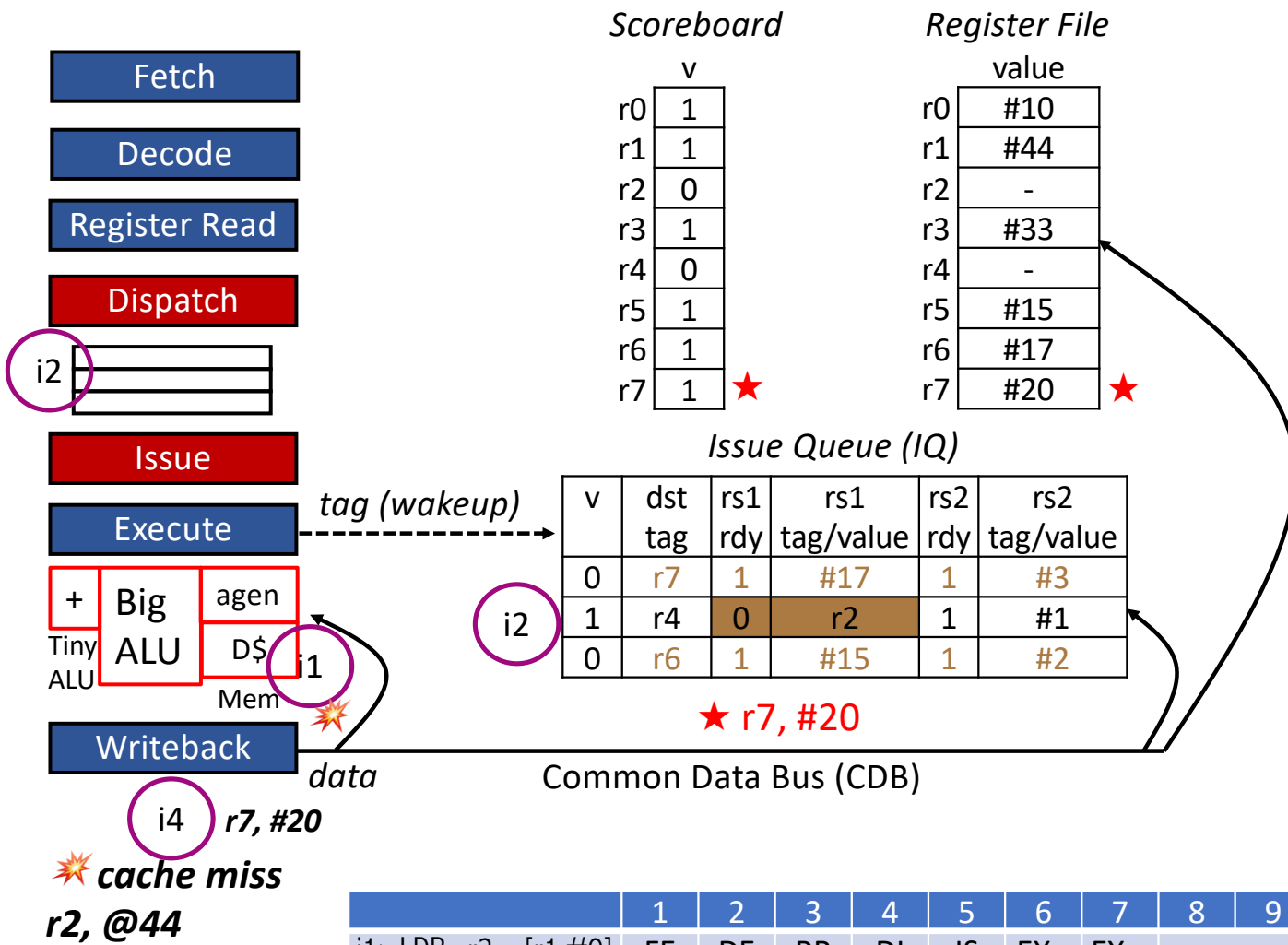
- **Dispath stage**
 - Copy the instruction from the RR/DI PPR to the issue queue (if there is an empty slot in the queue)
 - Set **v** to 1 (means **busy**) for the occupied slot in the instruction queue
- **Issue stage**
 - If both operands are ready, the selection logic sends the instruction to the execution unit
 - Deallocate the issue queue entry by setting **v** = 0
- **Scoreboard**
 - When an instruction has register **rN** as a destination ($N = 0 - 7$), set the corresponding bit to 0 (**NOT READY**)
 - Instructions capture the tag if **v**=0 and value otherwise (from RF)

CDC 6600 Scoreboard (Key Additions)

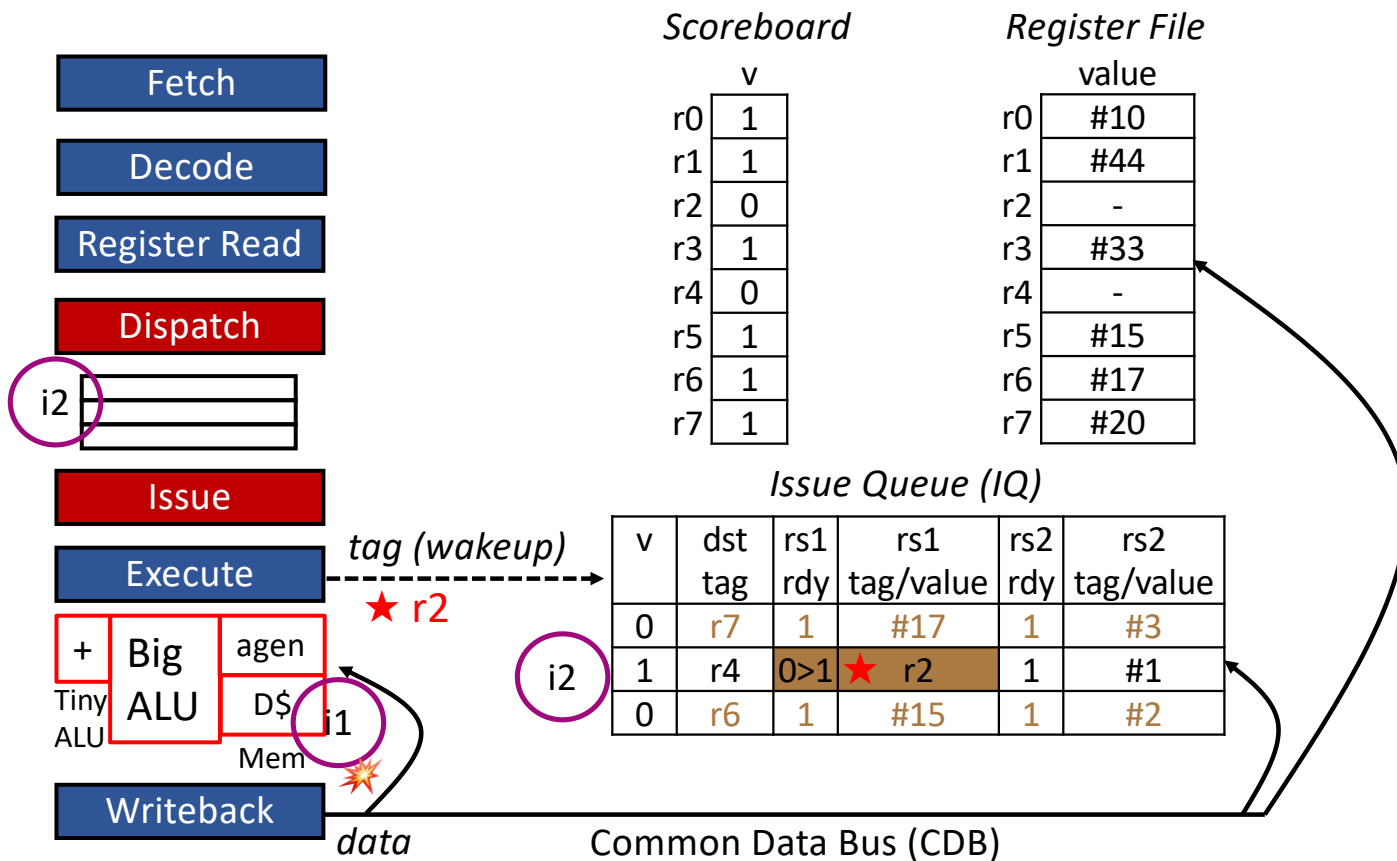
- **Instruction wakeup and select**
 - The *wakeup logic* in front of the issue queue *snoops* for destination tags of parent instructions. When the destination tag appears, it wakes up all instructions waiting for that tag.
 - X: tag, X+1: value, capture-tag-and-go
 - The *selection logic* in the issue stage decides which of the “ready” instructions to execute next.
- **Common Data Bus (CDB)**
 - The values are broadcasted over the common data bus bypassing register file writes. This bus resembles the forwarding/bypass network in the MIPS pipeline



	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
i1: LDR r2, [r1, #0]	FE	DE	RR	DI	IS	EX@	EXD\$... miss ...									
i2: ADD r4, r2, #1		FE	DE	RR	DI	IS	IS	IS	IS								
i3: ADD r6, r5, #2			FE	DE	RR	DI	IS	EX	WB								
i4: ADD r7, r6, #3				FE	DE	RR	DI	IS	EX								

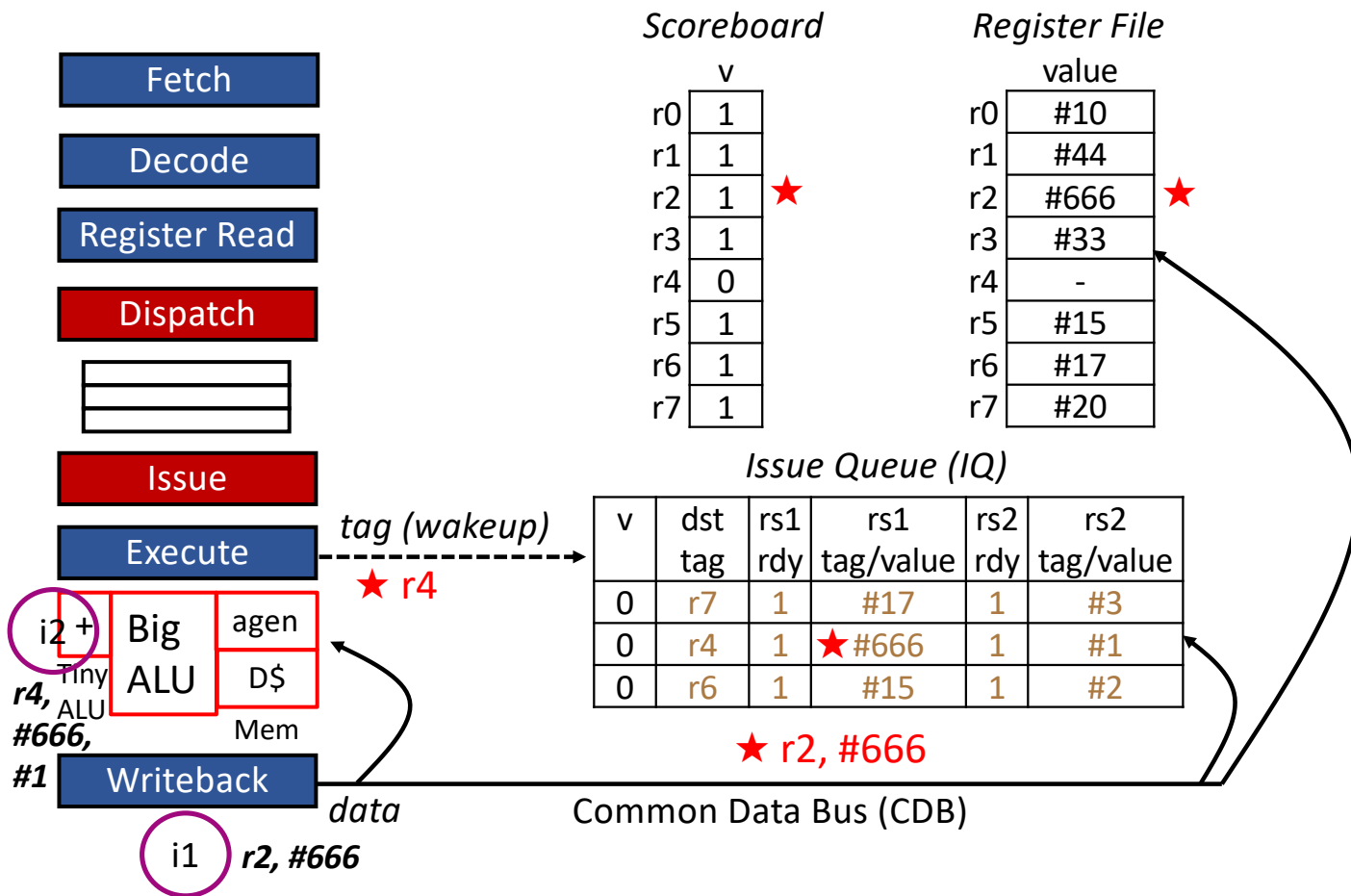


	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
i1: LDR r2, [r1,#0]	FE	DE	RR	DI	IS	EX@	EXD\$... miss ...									
i2: ADD r4, r2, #1		FE	DE	RR	DI	IS	IS	IS	IS	IS							
i3: ADD r6, r5, #2			FE	DE	RR	DI	IS	EX	WB								
i4: ADD r7, r6, #3				FE	DE	RR	DI	IS	EX	WB							

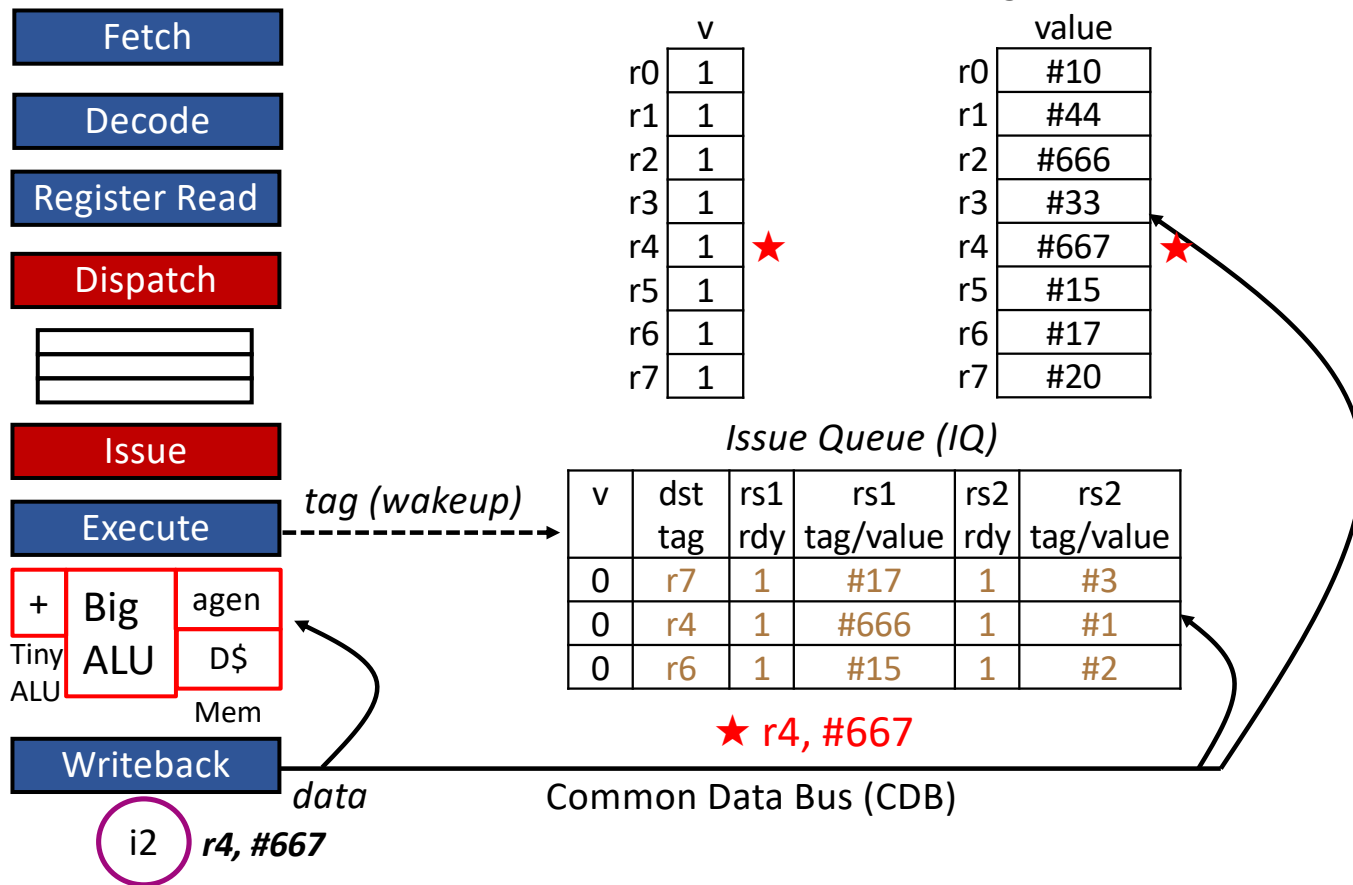


★ **cache miss**
r2, @44

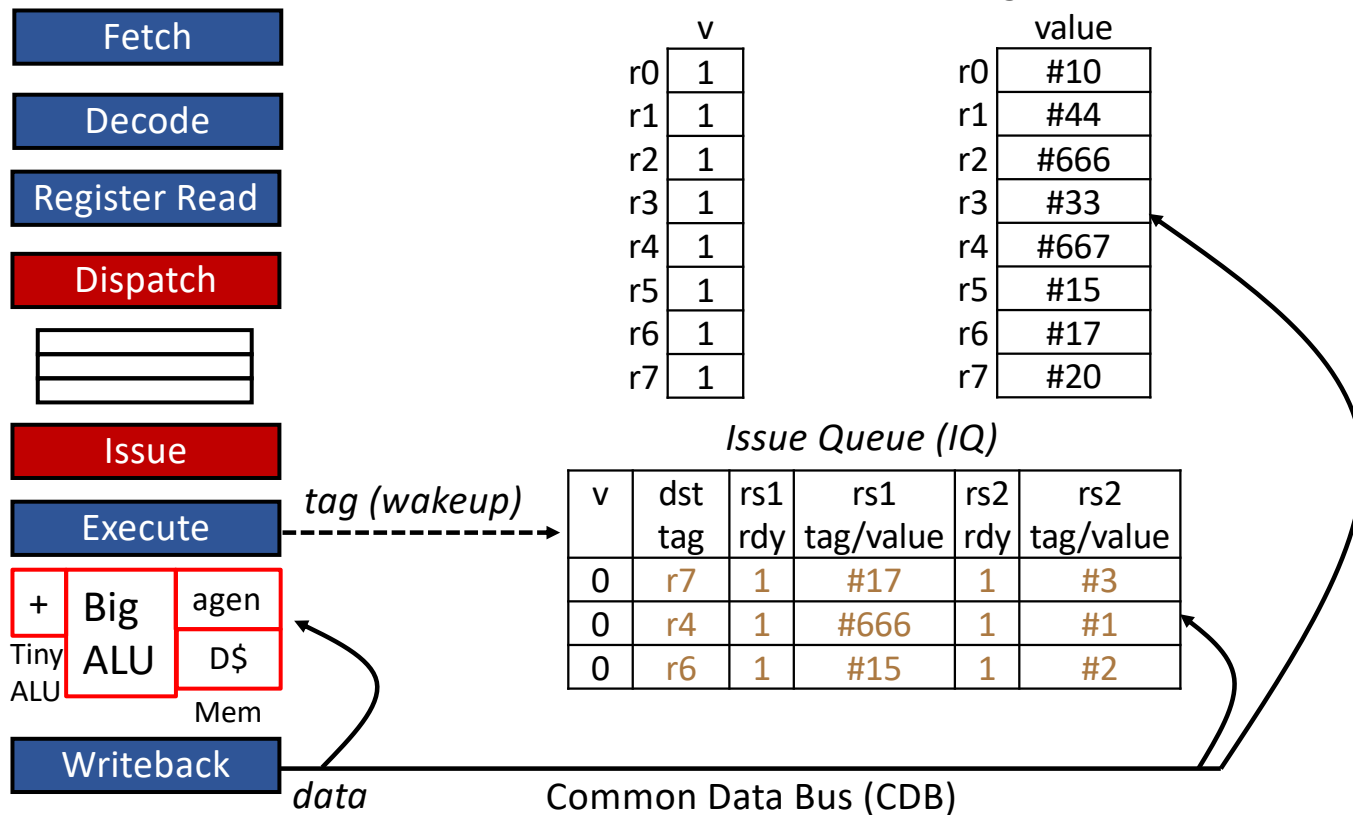
	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
i1: LDR r2, [r1,#0]	FE	DE	RR	DI	IS	EX@	EX _{D\$}	... miss ...									
i2: ADD r4, r2, #1		FE	DE	RR	DI	IS	IS	IS	IS	IS	IS						
i3: ADD r6, r5, #2			FE	DE	RR	DI	IS	EX	WB								
i4: ADD r7, r6, #3				FE	DE	RR	DI	IS	EX	WB							



	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
i1: LDR r2, [r1,#0]	FE	DE	RR	DI	IS	EX@	EX _{D\$}	... miss ...				WB					
i2: ADD r4, r2, #1		FE	DE	RR	DI	IS	IS	IS	IS	IS	IS	EX					
i3: ADD r6, r5, #2			FE	DE	RR	DI	IS	EX	WB								
i4: ADD r7, r6, #3				FE	DE	RR	DI	IS	EX	WB							



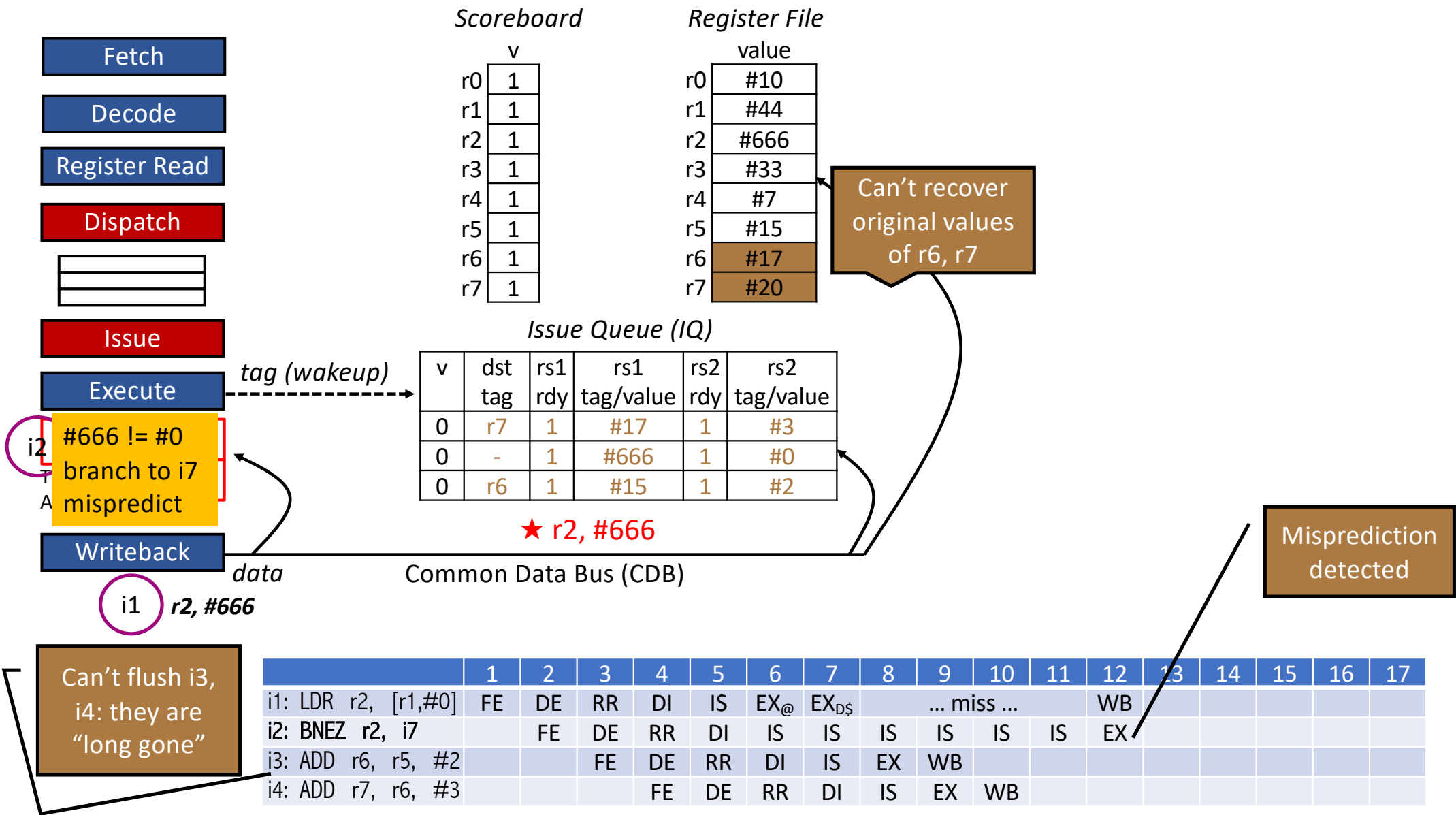
	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
i1: LDR r2, [r1,#0]	FE	DE	RR	DI	IS	EX@	EX _{D\$}	... miss ...				WB					
i2: ADD r4, r2, #1		FE	DE	RR	DI	IS	IS	IS	IS	IS	IS	EX	WB				
i3: ADD r6, r5, #2			FE	DE	RR	DI	IS	EX	WB								
i4: ADD r7, r6, #3				FE	DE	RR	DI	IS	EX	WB							



	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
i1: LDR r2, [r1,#0]	FE	DE	RR	DI	IS	EX@	EX _{D\$}	... miss ...				WB					
i2: ADD r4, r2, #1		FE	DE	RR	DI	IS	IS	IS	IS	IS	IS	EX	WB				
i3: ADD r6, r5, #2			FE	DE	RR	DI	IS	EX	WB								
i4: ADD r7, r6, #3				FE	DE	RR	DI	IS	EX	WB							

Two problems with OOO v.1

- Cannot recover from misspeculation (cannot schedule past a basic block)
 - Younger instructions are **speculative** with respect to older instructions
 - Possible to have **older predicted branches** that have not executed yet
 - Older load instructions may have executed speculatively with respect to prior unresolved stores
- Exceptions are not **precise**, i.e., register file is being updated out of the original program order
- Reverts to in-order when two producers have the same destination register
 - **WAR** and **WAW** lead to stalls
 - Must stall younger producer in Register Read stage until older producer executes



We were Here.

Precise Interrupts

Precise Interrupts

- A interrupt is precise if
 - all instructions prior to the interrupt-generating instructions update the architectural state in program order
 - the instruction generating the interrupt updates the architectural state in the same order (on an out-of-order machine) as it would on a single-cycle in-order machine
- To implement precise interrupts in a dynamically scheduled machine, we need a way to execute instructions OOO, but make them update the architectural state in program order

Hardware Speculation

Hardware Speculation

- Combines four key ideas
 - Register renaming to avoid WAR and WAW hazards
 - Dynamic branch prediction to avoid control hazards
 - Dynamic scheduling to execute instruction OOO
 - Reorder buffer (ROB) for precise interrupts and recovery from misspeculation

Two Humps in a Modern Pipeline

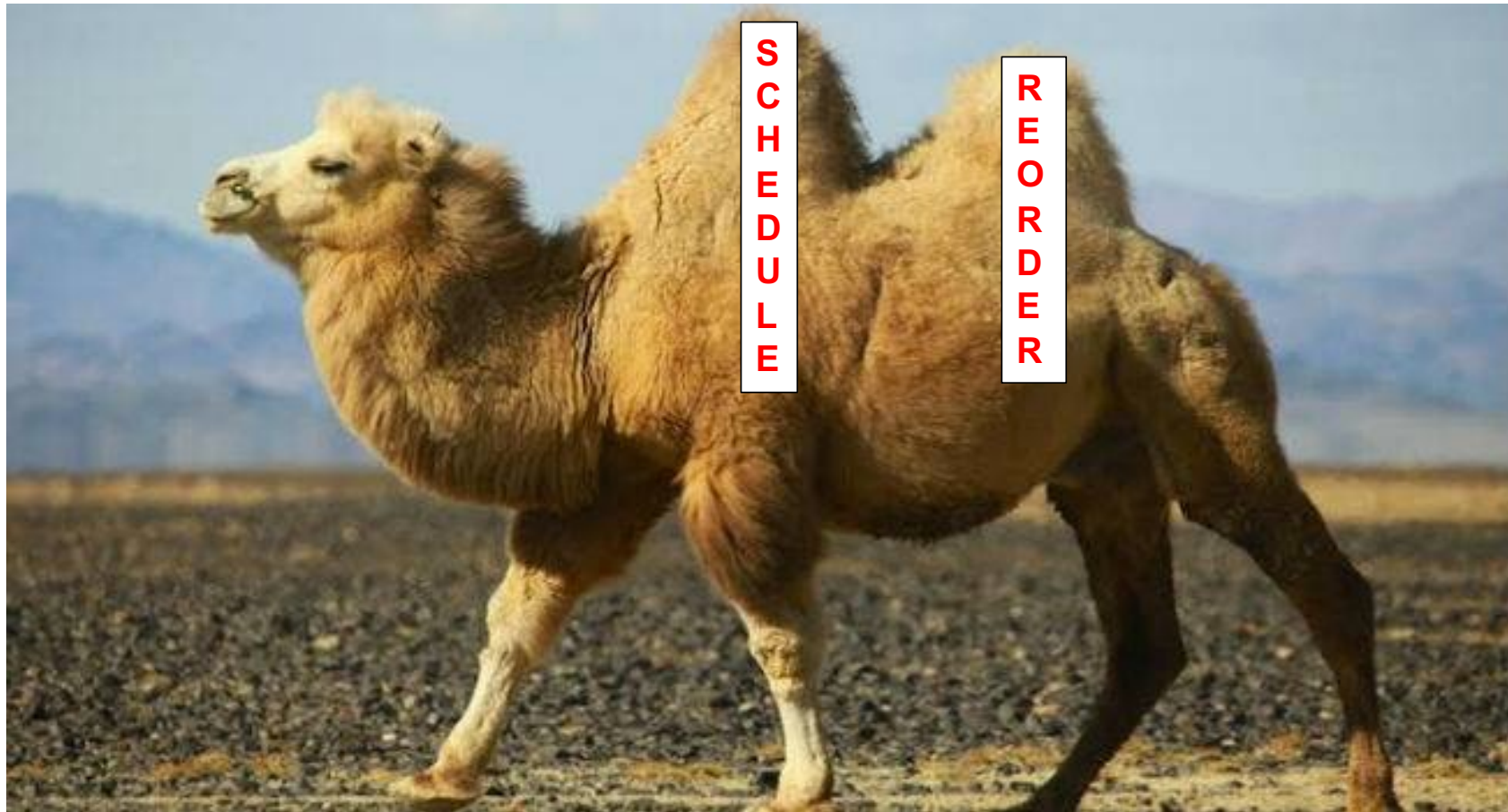


Photo credit: <http://true-wildlife.blogspot.ch/2010/10/bactrian-camel.html>

Out-of-Order Pipeline (v.2)

- **Solution: Reorder Buffer (ROB)**
 - ROB enables OOO execution, while at the same time supports recovery from mispredictions and exceptions
 - ROB also implements **register renaming**
 - Rename non-unique destination tags (**architectural register specifiers**) to unique destination tags (**ROB tags**)
 - Source tags are renamed as well, linking without ambiguity consumers to their producers
 - No reverting back to in-order due to WAR and WAW hazards, as they are eliminated after renaming

Operation with ROB (1-Page Cheat sheet)

The “Register File” is replaced with an expanded set of registers split into two parts

- **Architectural Register File (ARF):** Contains values of architectural registers as if produced by an in-order pipeline. That is, contains committed (non-speculative) versions of architectural registers to which the pipeline may safely revert to if there is a misprediction or exception.
- **Reorder Buffer (ROB):** Contains speculative versions of architectural registers. There may be multiple speculative versions for a given architectural register.

ROB is a circular FIFO with head and tail pointers

- A list of oldest to youngest instructions in program order
- Instruction at ROB Head is oldest instruction
- Instruction at ROB Tail is youngest instruction

New **Rename Stage** (after Decode and before Register Read)

- The new instruction is allocated to the ROB entry pointed to by ROB Tail. This is also its unique “ROB tag”.
- Source register specifiers are renamed to the expanded set of registers, the ARF+ROB. Renaming pinpoints the location of the value: ARF or ROB, and where in the ROB (ROB tag of producer). Thus, renaming unambiguously links consumers to their producers.
- Destination register specifier is renamed to the instruction’s unique ROB tag.
- **Rename Map Table (RMT)** contains the bookkeeping for renaming. (Intel calls it the Register Alias Table (RAT).)

Register Read Stage

- Obtain source value from ARF or ROB (using renamed source)
- If renamed to ROB, ROB may indicate value not ready yet
 - Producer hasn’t executed yet
 - Keep renamed source as proxy for value
- A consumer instruction obtains its source values from ARF, ROB, and/or bypass, depending on situation:
 - ARF: if producer of value has retired from ROB
 - ROB: if producer of value has executed but not yet retired from ROB
 - Bypass: if producer of value has not yet executed

Writeback Stage

- Instruction writes its speculative result OOO into ROB instead of ARF (at its ROB entry)

New **Retire** Stage safely commits results from ROB to ARF in program order

Misprediction/exception recovery

- Offending instruction posts misprediction or exception bit in its ROB entry OOO
- Wait until offending instruction reaches head of ROB (oldest unretired instruction)
- Squash all instructions in pipeline and ROB, and restore RMT to be consistent with an empty pipeline

Expanded Registers

- The “Register File” is replaced with an expanded set of registers split into two parts
 - **Architectural Register File (ARF):** Contains values of architectural registers as if produced by an in-order pipeline. That is, contains committed (non-speculative) versions of architectural registers to which the pipeline may safely revert to if there is a misprediction or exception.
 - **Reorder Buffer (ROB):** Contains speculative versions of architectural registers. There may be multiple speculative versions for a given architectural register.

Register Renaming

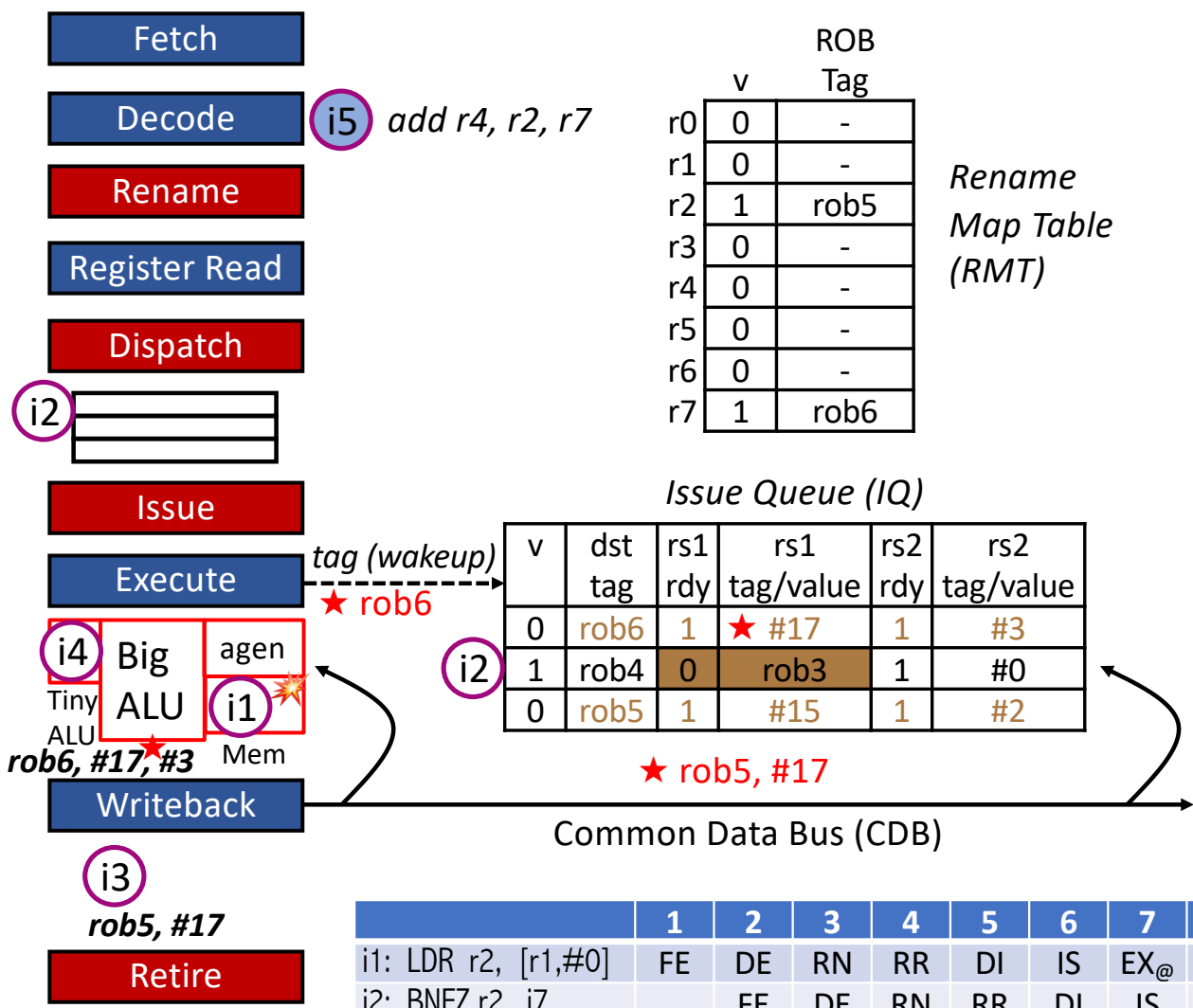
- New **Rename Stage** (after Decode and before Register Read)
 - The new instruction is allocated to the ROB entry pointed to by ROB Tail. This is also its unique “ROB tag”
 - Source register specifiers are renamed to the expanded set of registers, the ARF+ROB. Renaming pinpoints the location of the value: ARF or ROB, and where in the ROB (ROB tag of producer). Thus, renaming unambiguously links consumers to their producers.
 - Destination register specifier is renamed to the instruction’s unique ROB tag.
 - **Rename Map Table (RMT)** contains the book-keeping for renaming. (Intel calls it the Register Alias Table (RAT))

Register Renaming

- **Register Read Stage**
 - Obtain source value from ARF or ROB (using renamed source)
 - If **renamed to ROB**, ROB may indicate value not ready yet
 - Producer hasn't executed yet
 - Keep renamed source as proxy for value
 - A consumer instruction obtains its source values from ARF, ROB, and/or bypass, depending on situation:
 - **ARF**: if producer of value has retired from ROB
 - **ROB**: if producer of value has executed but not yet retired from ROB
 - **Bypass**: if producer of value has not yet executed

Writeback, Retirement, and Recovery

- Writeback Stage
 - Instruction writes its **speculative result** OOO into ROB instead of ARF (at its ROB entry)
- New **Retire** Stage safely **commits** results from **ROB to ARF** in program order
- **Misprediction/exception recovery**
 - Offending instruction posts misprediction or exception bit in its ROB entry OOO
 - Wait until **offending instruction** reaches head of ROB (oldest unretired instruction)
 - **Squash all instructions** in pipeline and ROB, and restore RMT to be consistent with an empty pipeline



value

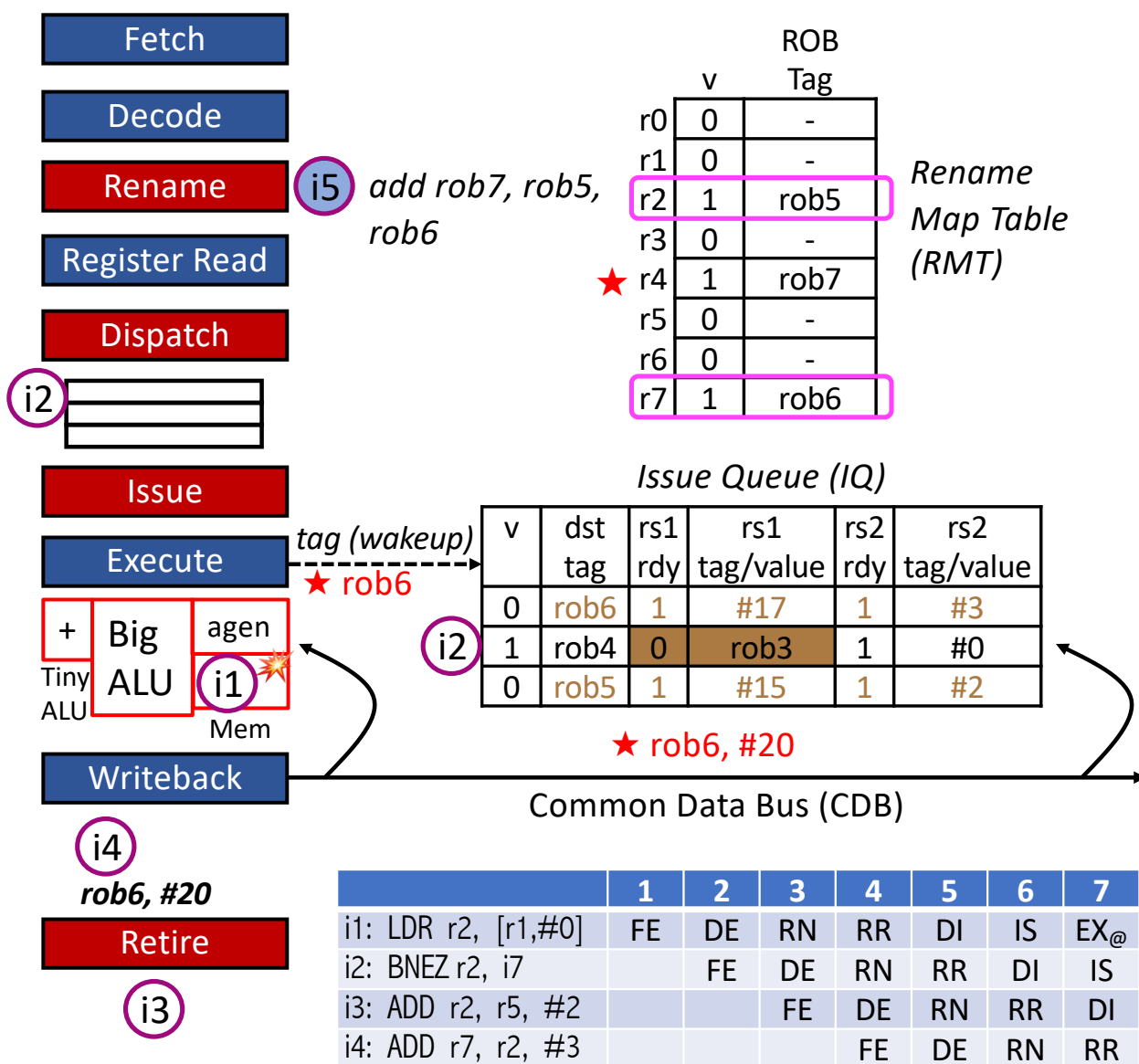
r0	#10
r1	#44
r2	#11
r3	#33
r4	#7
r5	#15
r6	#-7
r7	#345

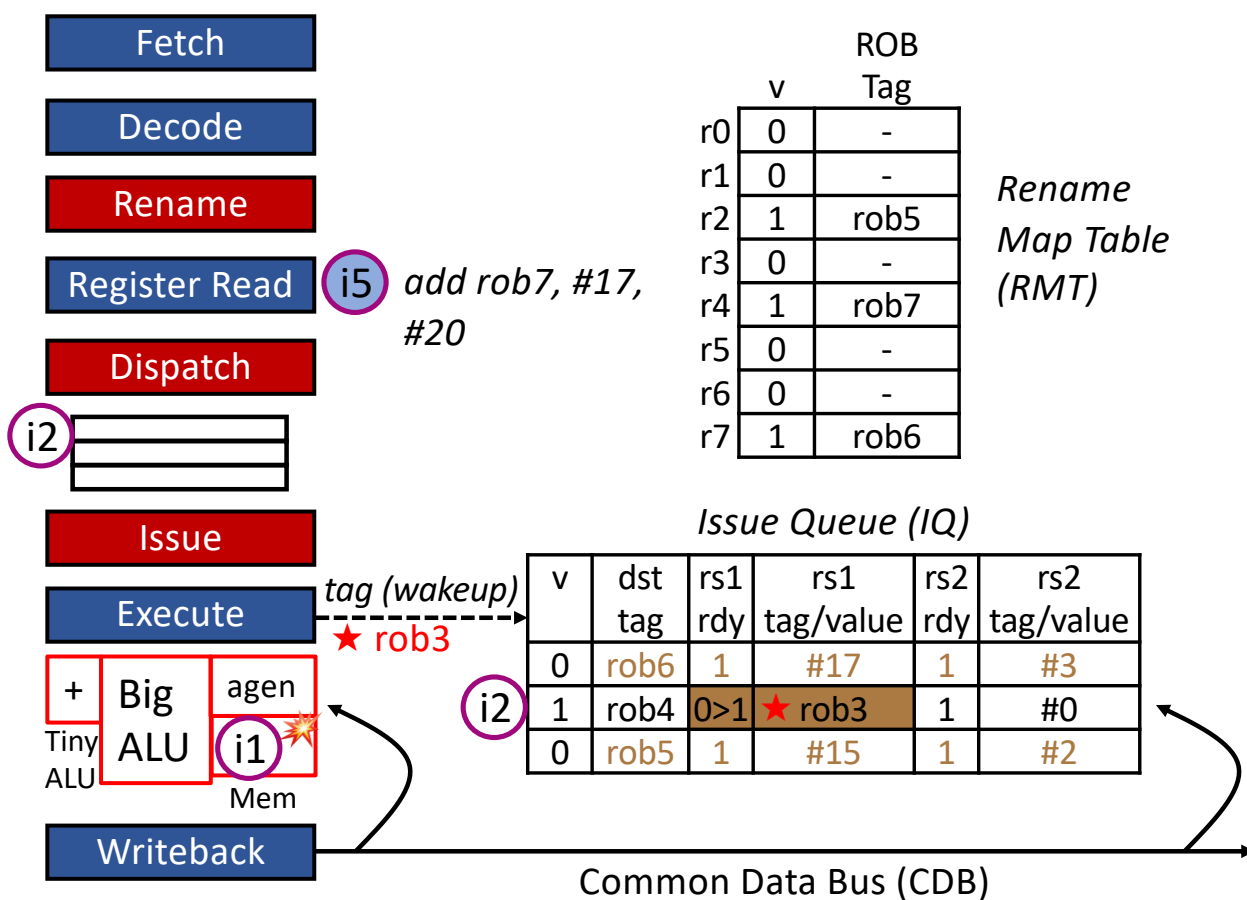
Architectural Register File (ARF)

value

	value	dst	rdy	exc	misp	PC
rob0						
rob1						
rob2						
H rob3	-	r2	0	0	0	i1
rob4	-	-	0	0	0	i2
★ rob5	#17	r2	1	0	0	i3
rob6	-	r7	0	0	0	i4
T rob7						
...						
rob31						

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
i1: LDR r2, [r1,#0]	FE	DE	RN	RR	DI	IS	EX@	EXD\$...miss...								
i2: BNEZ r2, i7		FE	DE	RN	RR	DI	IS	IS	IS	IS							
i3: ADD r2, r5, #2			FE	DE	RN	RR	DI	IS	EX	WB							
i4: ADD r7, r2, #3				FE	DE	RN	RR	DI	IS	EX							





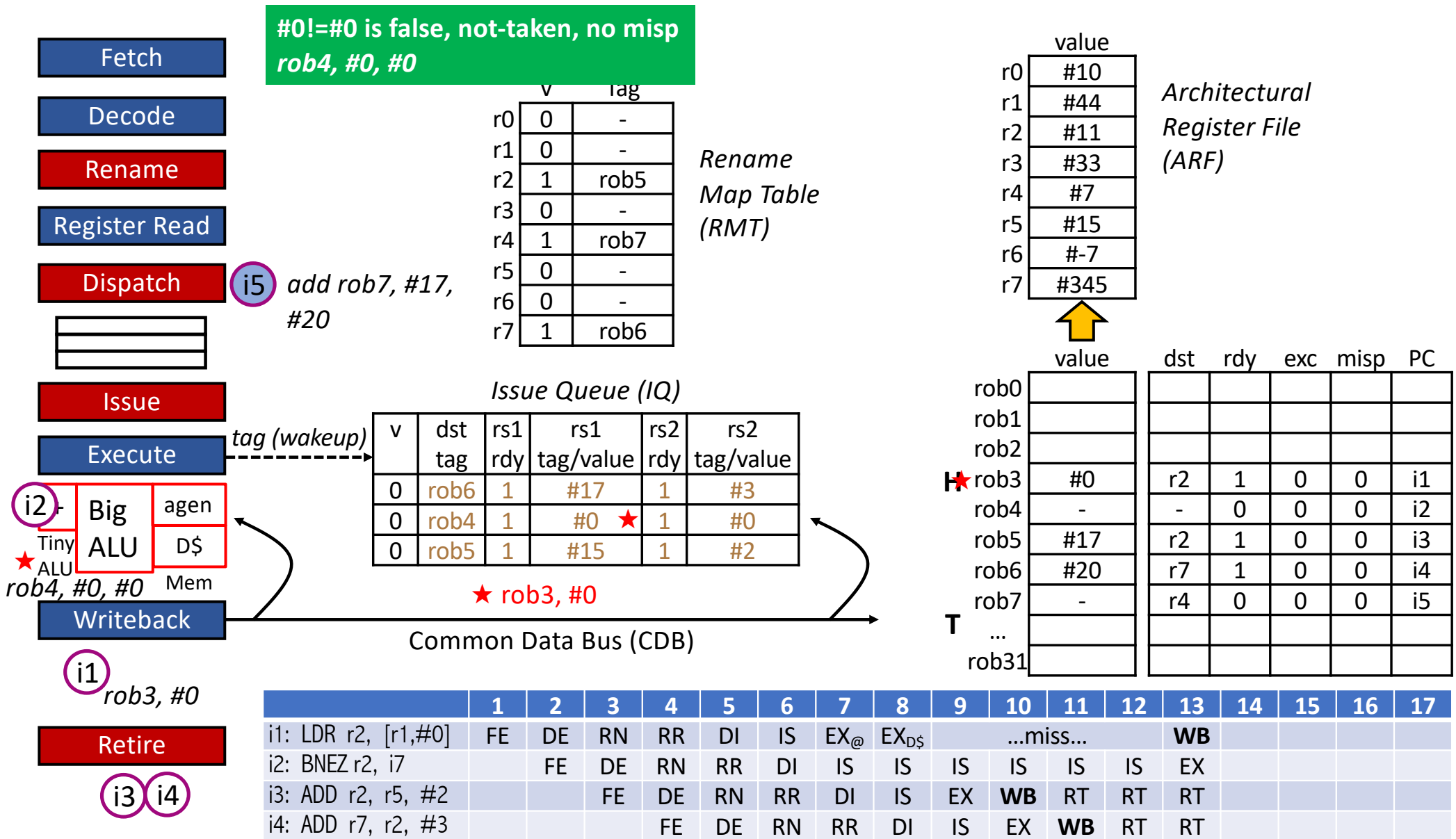
Architectural Register File (ARF)

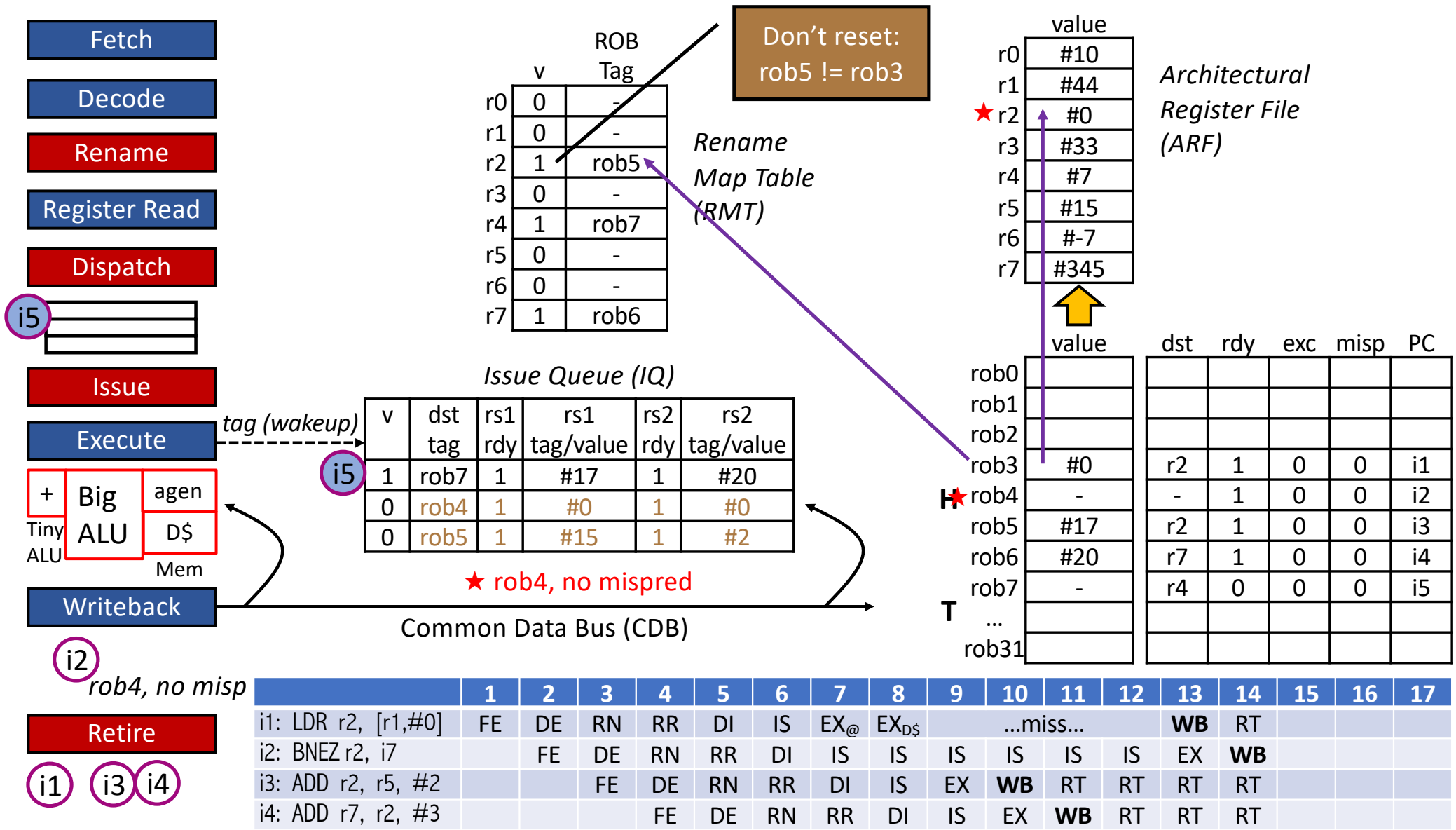
	value
r0	#10
r1	#44
r2	#11
r3	#33
r4	#7
r5	#15
r6	#-7
r7	#345

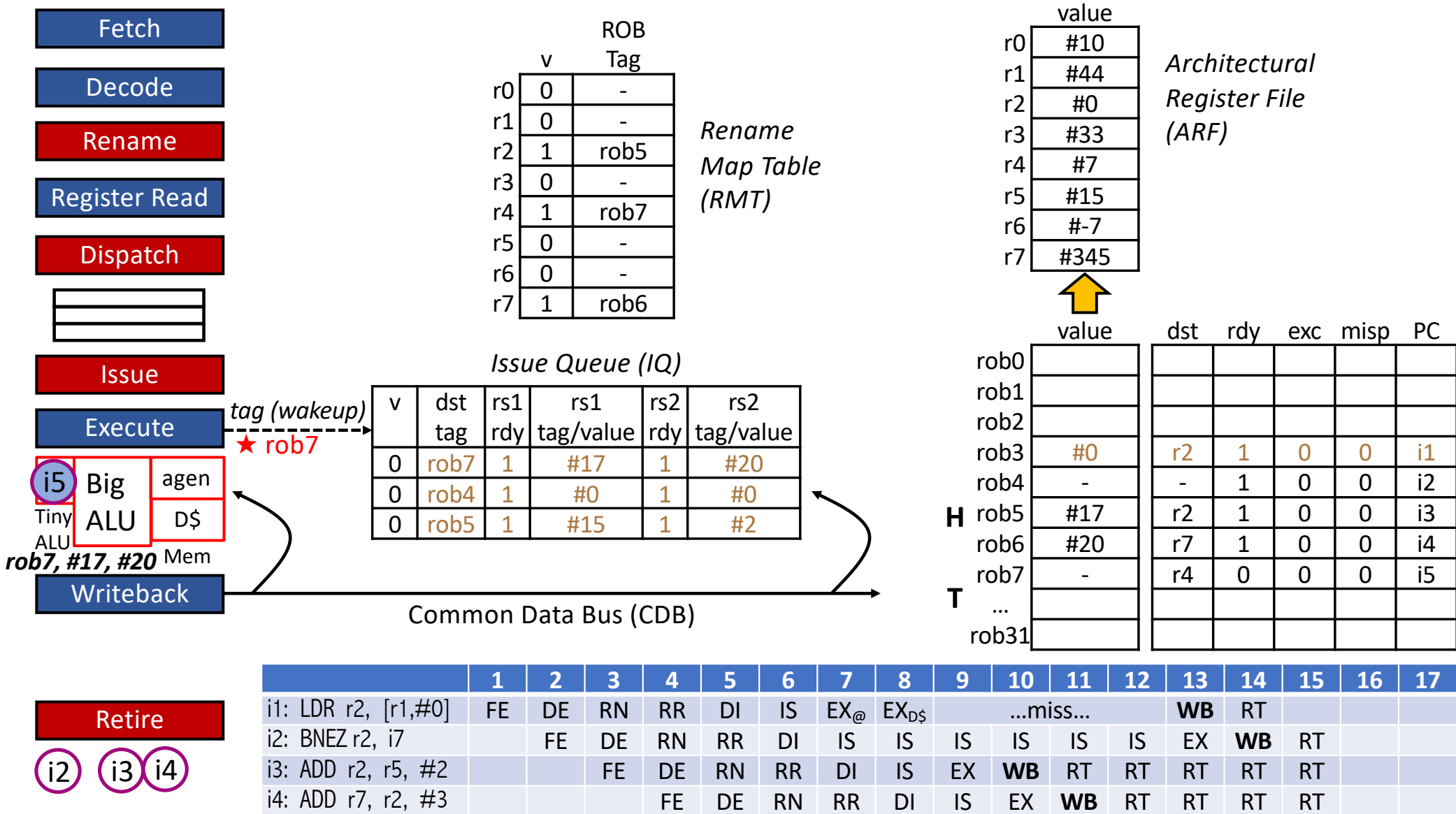
Issue Queue (IQ) Table:

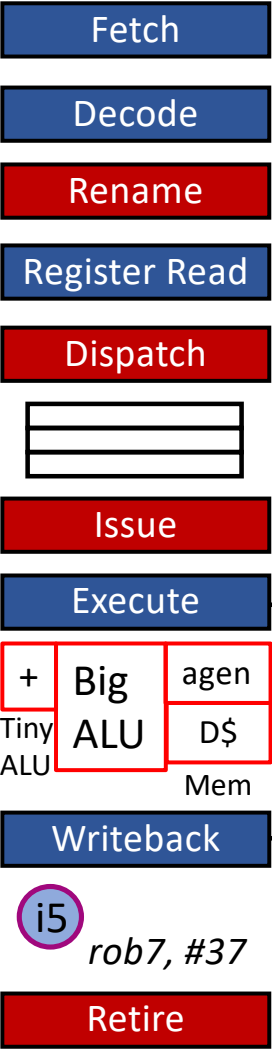
	value	dst	rdy	exc	misp	PC
rob0						
rob1						
rob2						
H rob3	-	r2	0	0	0	i1
rob4	-	-	0	0	0	i2
rob5	#17	r2	1	0	0	i3
rob6	#20	r7	1	0	0	i4
rob7	-	r4	0	0	0	i5
T ...						
rob31						

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
i1: LDR r2, [r1,#0]	FE	DE	RN	RR	DI	IS	EX _@	EX _{D\$}	...miss...								
i2: BNEZ r2, i7		FE	DE	RN	RR	DI	IS	IS	IS	IS	IS	IS					
i3: ADD r2, r5, #2			FE	DE	RN	RR	DI	IS	EX	WB	RT	RT					
i4: ADD r7, r2, #3				FE	DE	RN	RR	DI	IS	EX	WB	RT					









		ROB	
		v	Tag
r0	0	-	
r1	0	-	
★ r2	0	rob5	
r3	0	-	
r4	1	rob7	
r5	0	-	
r6	0	-	
r7	1	rob6	

Reset:
rob5 == rob5

Rename
Map Table
(RMT)

	value
r0	#10
r1	#44
★ r2	#17
r3	#33
r4	#7
r5	#15
r6	#-7
r7	#345

Architectural
Register File
(ARF)

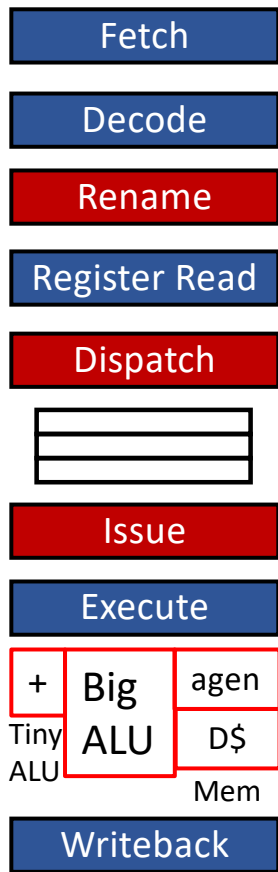
Issue Queue (IQ)					
v	dst tag	rs1 rdy	rs1 tag/value	rs2 rdy	rs2 tag/value
0	rob7	1	#17	1	#20
0	rob4	1	#0	1	#0
0	rob5	1	#15	1	#2

★ rob7, #37

	value	dst	rdy	exc	misp	PC
rob0						
rob1						
rob2						
rob3	#0	r2	1	0	0	i1
rob4	-	-	1	0	0	i2
rob5	#17	r2	1	0	0	i3
rob6	#20	r7	1	0	0	i4
★ rob7	#37	r4	0	0	0	i5
...						
rob31						

Common Data Bus (CDB)

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
i1: LDR r2, [r1,#0]	FE	DE	RN	RR	DI	IS	EX@	EXD\$...miss...			WB	RT			
i2: BNEZ r2, i7		FE	DE	RN	RR	DI	IS	IS	IS	IS	IS	IS	EX	WB	RT		
i3: ADD r2, r5, #2			FE	DE	RN	RR	DI	IS	EX	WB	RT	RT	RT	RT	RT	RT	
i4: ADD r7, r2, #3				FE	DE	RN	RR	DI	IS	EX	WB	RT	RT	RT	RT	RT	



Reset:
rob7 == rob7

	v	ROB Tag
r0	0	-
r1	0	-
r2	0	-
r3	0	-
★ r4	0	rob7
r5	0	-
r6	0	-
r7	0	-

Rename Map Table (RMT)

	value
r0	#10
r1	#44
r2	#17
r3	#33
★ r4	#37
r5	#15
r6	#-7
r7	#20

Architectural Register File (ARF)

Issue Queue (IQ)

v	dst tag	rs1 rdy	rs1 tag/value	rs2 rdy	rs2 tag/value
0	rob7	1	#17	1	#20
0	rob4	1	#0	1	#0
0	rob5	1	#15	1	#2

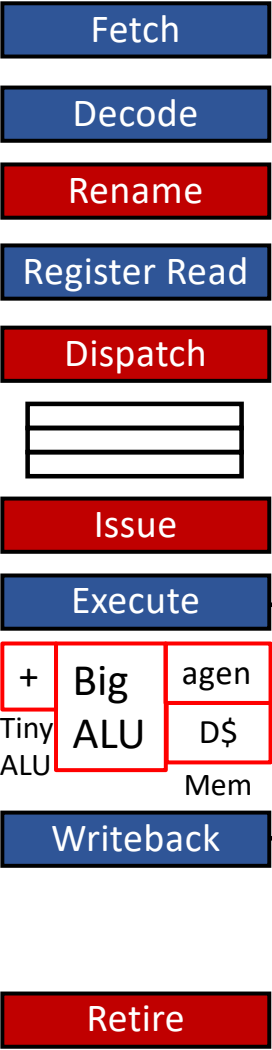
	value	dst	rdy	exc	misp	PC
rob0						
rob1						
rob2						
rob3	#0	r2	1	0	0	i1
rob4	-	-	1	0	0	i2
rob5	#17	r2	1	0	0	i3
rob6	#20	r7	1	0	0	i4
rob7	#37	r4	0	0	0	i5
HT ...						
rob31						

Common Data Bus (CDB)



i5

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
i1: LDR r2, [r1,#0]	FE	DE	RN	RR	DI	IS	EX _@	EX _{D\$}	...miss...				WB	RT			
i2: BNEZ r2, i7		FE	DE	RN	RR	DI	IS	IS	IS	IS	IS	IS	EX	WB	RT		
i3: ADD r2, r5, #2			FE	DE	RN	RR	DI	IS	EX	WB	RT	RT	RT	RT	RT	RT	
i4: ADD r7, r2, #3				FE	DE	RN	RR	DI	IS	EX	WB	RT	RT	RT	RT	RT	RT



ROB		
	v	Tag
r0	0	-
r1	0	-
r2	0	-
r3	0	-
r4	0	-
r5	0	-
r6	0	-
r7	0	-

Rename Map Table (RMT)

Issue Queue (IQ)

v	dst tag	rs1 rdy	rs1 tag/value	rs2 rdy	rs2 tag/value
0	rob7	1	#17	1	#20
0	rob4	1	#0	1	#0
0	rob5	1	#15	1	#2

Architectural Register File (ARF)					
	value				
r0	#10				
r1	#44				
r2	#17				
r3	#33				
r4	#37				
r5	#15				
r6	#-7				
r7	#20				

↑

	value	dst	rdy	exc	misp	PC
rob0						
rob1						
rob2						
rob3	#0	r2	1	0	0	i1
rob4	-	-	1	0	0	i2
rob5	#17	r2	1	0	0	i3
rob6	#20	r7	1	0	0	i4
rob7	#37	r4	1	0	0	i5
HT ...						
rob31						

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
i1: LDR r2, [r1,#0]	FE	DE	RN	RR	DI	IS	EX@	EXD\$...miss...				WB	RT			
i2: BNEZ r2, i7		FE	DE	RN	RR	DI	IS	IS	IS	IS	IS	IS	EX	WB	RT		
i3: ADD r2, r5, #2			FE	DE	RN	RR	DI	IS	EX	WB	RT	RT	RT	RT	RT	RT	
i4: ADD r7, r2, #3				FE	DE	RN	RR	DI	IS	EX	WB	RT	RT	RT	RT	RT	RT

Cycle # 1

❖ i1 (Fetch)

Cycle # 2

❖ i1 (Decode)

❖ i2 (Fetch)

Cycle # 3

❖ i_1 (Rename)

1. Allocate entry for i_1 in ROB at rob_3
 - ❖ Tail of ROB is at rob_3
2. Rename the destination operand (r_2) to rob_3
3. Increment the tail pointer of ROB to rob_4
4. Set $v[r_2] = 1$ in RMT
5. One source operand is a constant 0
6. Rename the second source operand r_1 to $ARF[r_1]$ because
in RMT: $v[r_1] = 0$

❖ i_2 (Decode)

❖ i_3 (Fetch)

- ❖ The fetch is speculative as i_2 is a branch and it may be taken
(our branch prediction strategy is [always-untaken](#))

Cycle # 4

❖ i1 (Register Read)

1. Read the value of the second source operand from the register file: $ARF[r1]$ is 44

❖ i2 (Rename)

1. Allocate an entry for i2 in ROB at rob4
2. Rename the destination r2 to rob4
3. Move ROB tail to rob5
4. Rename the source operand r2 to rob3 because in RMT:
 $v[r2]=1$
 - ❖ Carry this tag to the issue queue (later) and wait for the value to be produced by the producer (i1)

❖ i3 (Decode)

❖ i4 (Fetch)

Cycle # 5

- ❖ i1 (Dispatch)
 - 1. Instruction is being copied into the issue queue
 - ❖ There are free entries in the issue queue
- ❖ i2 (Register Read)
 - 1. Nothing to read from register file (source operand is not ready)
- ❖ i3 (Rename)
 - 1. Allocate an entry for i3 in ROB at rob5
 - 2. Rename the destination r2 to rob5, keep $v[r2]=1$ in RMT
 - 3. Move ROB tail to rob6
 - 4. Rename the source operand r5 to ARF[r5] because in RMT:
 $v[r5]=0$
- ❖ i4 (Decode)

Cycle # 6

❖ i1 (Issue)

1. Instruction is now inside the issue queue
 - ❖ $v=1$ to indicate the slot in the issue queue has been occupied
 - ❖ The scheduler will pick this instruction for execution (next cycle)
 - ❖ Source operands ready ($rs1\ rdy=1$ and $rs2\ rdy=1$)

❖ i2 (Dispatch)

1. Instruction is being copied into the issue queue

❖ i3 (Register Read)

1. Read $ARF[r5]=\#15$

❖ i4 (Rename)

1. Allocate an entry for i4 in ROB at rob6 (tail moves to r7)
2. Rename the destination r7 to rob6, set $v[r7]=1$ in RMT
3. Rename r2 to rob5 because in RMT: $v[r2]=1$

Cycle # 7

❖ i1 (Execute (Agen))

1. Instruction has been issued to the functional unit (agen) for address calculation: source operands are #0 and #44
2. The corresponding issue queue slot has been freed ($v=0$)

❖ i2 (Issue)

1. Instruction is now inside the issue queue
 - ❖ $v=1$ to indicate the slot in the issue queue has been occupied
 - ❖ The scheduler will pick this instruction for execution when both source operands are ready ($rs1\ rdy=0$)

❖ i3 (Dispatch)

1. Instruction is being copied into the issue queue

❖ i4 (Register Read)

1. Nothing to read from register file (source operand is not ready)

Cycle # 8

❖ i1 (Execute (D\$))

1. Instruction is checking the SRAM data cache for value

❖ i2 (Issue)

1. Instruction remains in the issue queue due to a **RAW** hazard

❖ i3 (Issue)

1. Instruction is now inside the issue queue

- ❖ $v=1$ to indicate the slot in the issue queue has been occupied

- ❖ The scheduler will pick this instruction for execution next cycle as source operands are ready ($rs1\ rdy=1$ and $rs2\ rdy=1$)

- ❖ ALU is free for executing another instruction

❖ i4 (Dispatch)

1. Instruction is being copied into the issue queue

Cycle # 9

- ❖ i1 (Execute(...miss...))
 1. Cache miss is being resolved (data being read from main memory)
- ❖ i2 (Issue)
 1. Instruction remains in the issue queue due to a RAW hazard
- ❖ i3 (Execute)
 1. Instruction is issued to the Tiny ALU (deallocated from issue queue)
 2. At the end of the cycle, the instruction send its destination tag (rob5) to the wakeup logic in front of the issue queue
- ❖ i4 (Issue)
 1. Instruction is now inside the issue queue (will execute next cycle)
 - ❖ v=1 to indicate the slot in the issue queue has been occupied
 - ❖ rs1 rdy changes from 0 to 1 as the wakeup logic has been notified of the availability of rob5; and rs2 rdy=1

Cycle # 10

- ❖ i1 (Execute(...miss...))
 1. Cache miss is being resolved (data being read from main memory)
- ❖ i2 (Issue)
 1. Instruction remains in the issue queue due to a RAW hazard
- ❖ i3 (Writeback)
 1. Instruction writes the result to its destination entry in the ROB (rob5)
 2. Broadcasts the tag/value over the CDB to forward it to waiting insts.
- ❖ i4 (Execute)
 1. Instruction is issued to the Tiny ALU (deallocated from issue queue)
 2. At the end of the cycle, the instruction sends its tag (rob6) to the wakeup logic

Cycle # 11

- ❖ `i1 (Execute(...miss...))`
 1. Cache miss is being resolved (data being read from main memory)
- ❖ `i2 (Issue)`
 1. Instruction remains in the issue queue due to a RAW hazard
- ❖ `i3 (Retire)`
 1. Instruction is waiting to reach the head of ROB to update the ARF with the value it has computed for `r2`
 2. Since older instructions haven't executed yet, and head of ROB is blocked, `i3` will wait for its turn to reach the head of ROB
- ❖ `i4 (Writeback)`
 1. Instruction writes the result to its destination entry in the ROB (`rob6`)
 2. Broadcasts the tag/value (`rob6, #20`) over the CDB to forward it to waiting insts.

Cycle # 12

- ❖ `i1 (Execute(...miss...))`
 1. Cache miss is resolved and instruction sends its dst. tag (`rob3`) to the issue queue waking up `i2`
- ❖ `i2 (Issue)`
 1. Instruction wakes up as its `rs1 rdy` changes from **0** to **1**
- ❖ `i3 (Retire)`
 1. Instruction is waiting to reach the head of ROB
- ❖ `i4 (Retire)`
 1. Instruction is waiting to reach the head of ROB to update the ARF with the value it has computed for `r7`
 2. Since older instructions haven't executed yet, and head of ROB is blocked, `i4` will wait for its turn to reach the head of ROB

Cycle # 13

❖ i1 (Writeback)

1. Instruction writes its result (**0**) to the `dst` entry in ROB at `rob3`

❖ i2 (Execute)

1. The branch condition is evaluated and there is no misprediction as the branch is (after execution) not taken
2. Instruction grabbed `r2` (renamed to `rob3`) from the CDB (forwarding)

❖ i3 (Retire)

1. Instruction is waiting to reach the head of ROB

❖ i4 (Retire)

1. Instruction is waiting to reach the head of ROB

Cycle # 14

❖ `i1` (Retire)

1. Instruction is at the head of ROB and in the retire stage
2. Updates `ARF[r2]` with the value it has in its entry on ROB
3. It checks the `ROB tag` in RMT and since tag corresponding to `r2` in RMT is not `rob3`, it leaves the `v` bit unchanged
4. Increment ROB head (moves to `rob4`)

❖ `i2` (Writeback)

1. No value to writeback as the instruction is a branch
2. Branch instruction sets the `misp` bit in ROB to 0 as the branch is not taken, and the prediction was that branch is not taken

❖ `i3` and `i4` (Retire)

1. Instructions are waiting to reach the head of ROB

Cycle # 15

- ❖ `i1` (`null`)
 - ❖ Instruction has retired (its gone!)
- ❖ `i2` (`Retire`)
 1. Nothing to write to ARF, so just retire from the pipeline
 2. Move head of ROB to `rob5`
- ❖ `i3` and `i4` (`Retire`)
 1. Instructions are waiting to reach the head of ROB

Cycle # 16

- ❖ `i1 (null)`
 - ❖ Instruction has retired (its gone!)
- ❖ `i2 (null)`
 - ❖ Instruction has retired (its gone!)
- ❖ `i3 (Retire)`
 1. Head of ROB so writes value (#17) to `ARF[r2]`
 2. It checks the `ROB tag` in RMT and since tag corresponding to `r2` in RMT is `rob5`, it resets the `v` bit to **0**
 3. Move head of ROB to `rob6`
- ❖ `i4 (Retire)`
 1. Instruction is waiting to reach the head of ROB

Cycle # 17

- ❖ `i1 (null)`
 - ❖ Instruction has retired (its gone!)
- ❖ `i2 (null)`
 - ❖ Instruction has retired (its gone!)
- ❖ `i3 (null)`
 - ❖ Instruction has retired (its gone!)
- ❖ `i4 (Retire)`
 1. Head of ROB so writes value (#20) to `ARF[r7]`
 2. It checks the `ROB tag` in `RMT` and since tag corresponding to `r7` in `RMT` is `rob67`, it resets the `v` bit to **0**
 3. Move head of ROB to `rob7`

Instruction i5

- ❖ Cycle #9 (Fetch)
 - ❖ Fetch is not blocked due to a branch and RAW hazard in the pipeline
- ❖ Cycle #10 (Decode)
- ❖ Cycle #11 (Rename)
 1. Allocate entry at `rob7` in ROB (increment the tail)
 2. Rename two source operands to `rob5` and `rob6` because `v[r2]` and `v[r7]` in RMT are 1
- ❖ Cycle #12 (Register Read)
 1. Both renamed src operands are available in the ROB. **Capture** the values
- ❖ Cycle #13 (Dispatch)
- ❖ Cycle # 14 (Issue) → Selected to execute next cycle
- ❖ Cycle # 15 (Execute) → Wakeup waiting instructions
- ❖ Cycle # 16 (Writeback)
- ❖ Cycle # 17-18 (Retire) → Head = Tail (Done!)

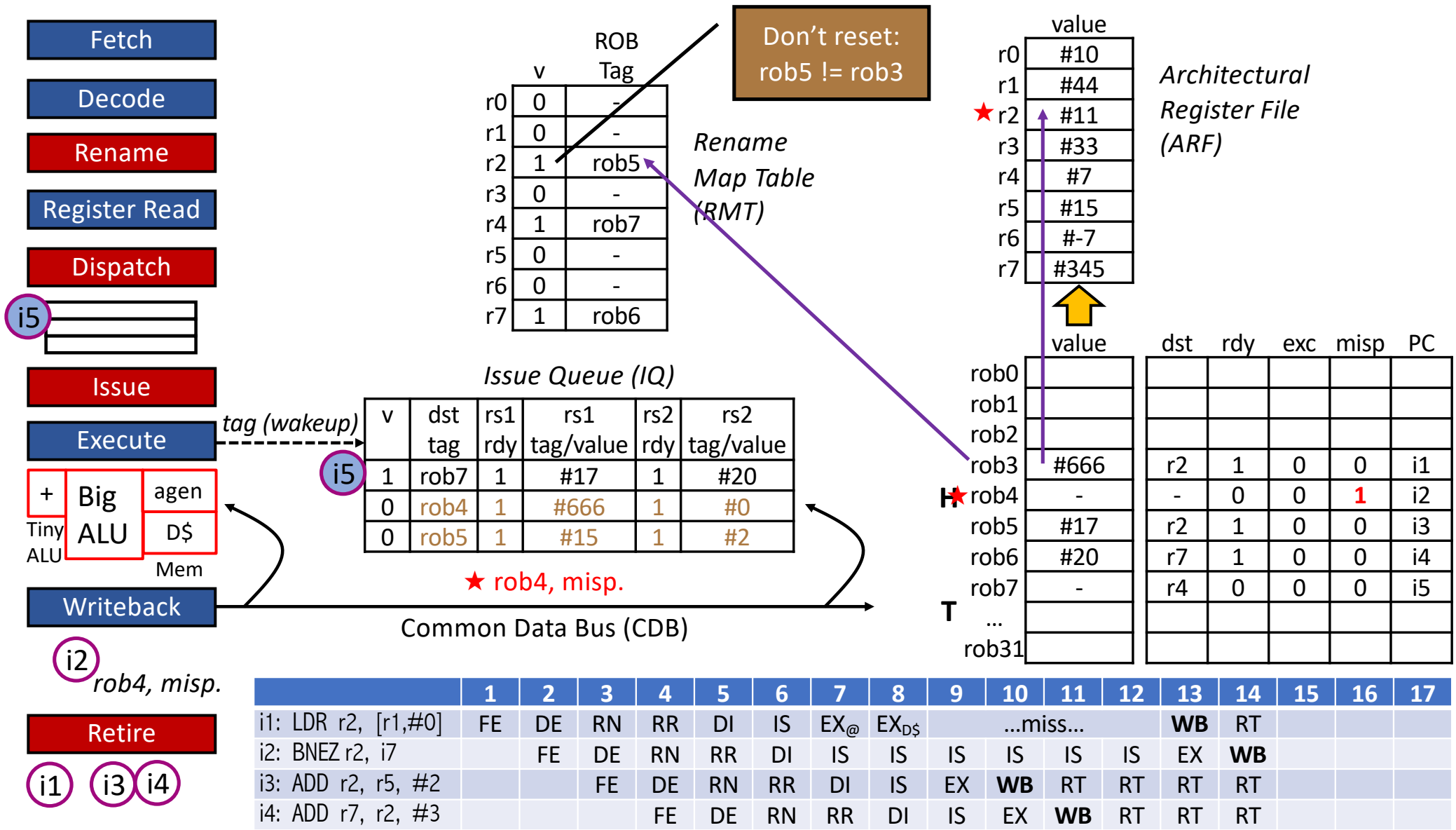
Observations

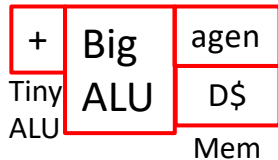
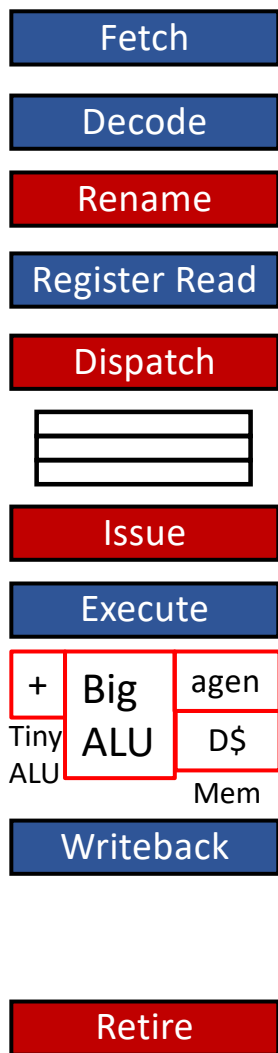
- **Compared to scoreboard only**
 - ROB did not **degrade** performance
 - Fetch **did not stall** as before (tolerated D\$ miss)
 - **In-order retirement** did not impede OOO, speculative execution
- **Recovery**
 - ROB was not called upon for recovery
 - But can see the danger of misprediction without ROB
 - Only leverages ROB for **renaming**

Recovery

Revise the previous scenario assuming mispredicted branch

- i1 (load instruction) gets the value #666 instead of #0





★ Recover RMT by flash-cleaning all valid bits

	v	Tag
r0	0	-
r1	0	-
r2	0	-
r3	0	-
r4	0	-
r5	0	-
r6	0	-
r7	0	-

Rename Map Table (RMT)

★ Squash ROB by setting T=H

	value
r0	#10
r1	#44
r2	#666
r3	#33
r4	#7
r5	#15
r6	#-7
r7	#345

Architectural Register File (ARF)

Issue Queue (IQ)

	v	dst tag	rs1 rdy	rs1 tag/value	rs2 rdy	rs2 tag/value
	0	rob7	1	#17	1	#20
	0	rob4	1	#666	1	#0
	0	rob5	1	#15	1	#2

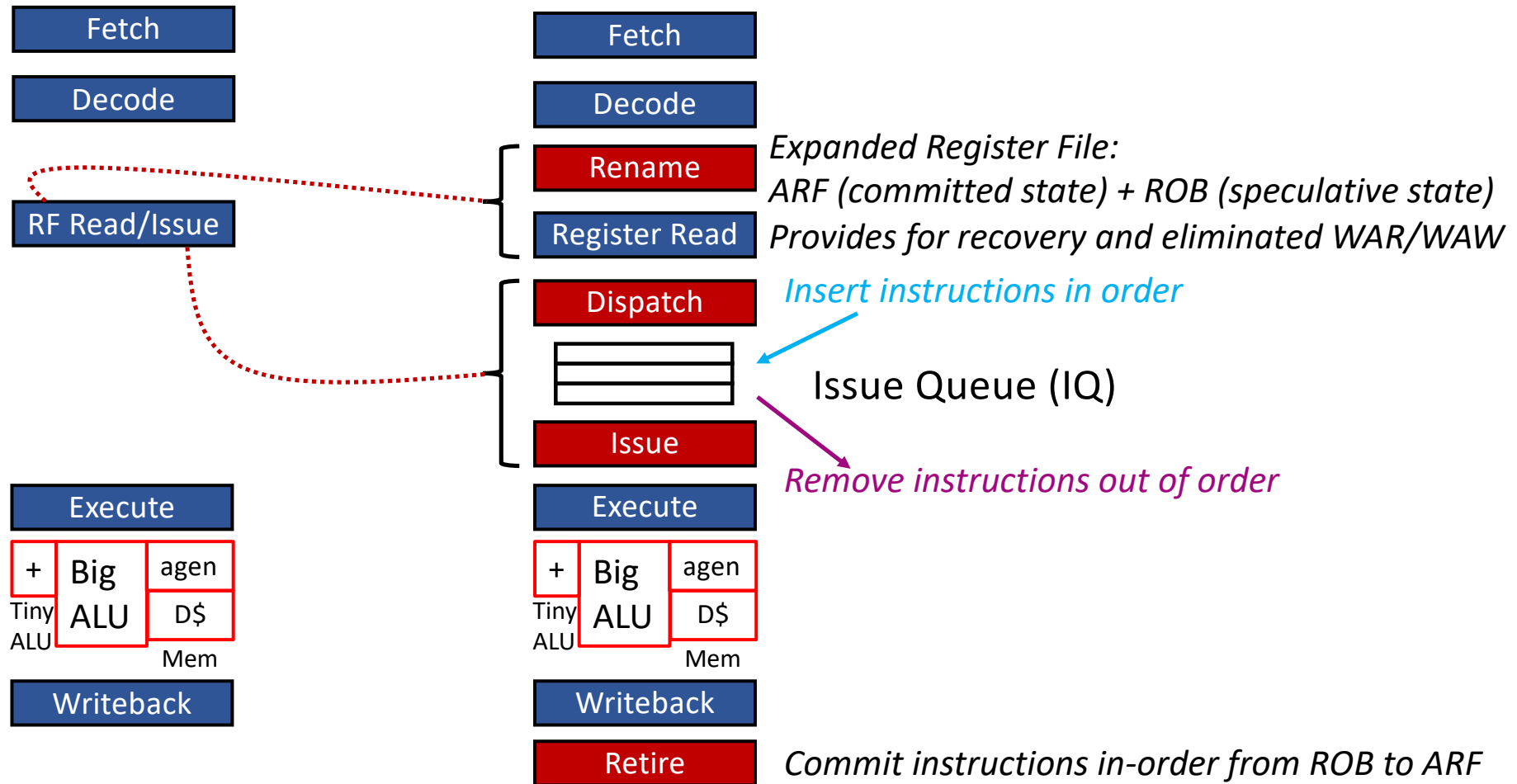
★ Squash all pipeline stages. (i5) They contain younger instructions than the ROB head, i.e., branch

	value	dst	rdy	exc	misp	PC
rob0						
rob1						
rob2						
rob3	#666	r2	1	0	0	i1
rob4	-	-	0	0	0	i2
rob5	#17	r2	1	0	0	i3
rob6	#20	r7	1	0	0	i4
rob7	-	r4	0	0	0	i5
...						
rob31						

i2

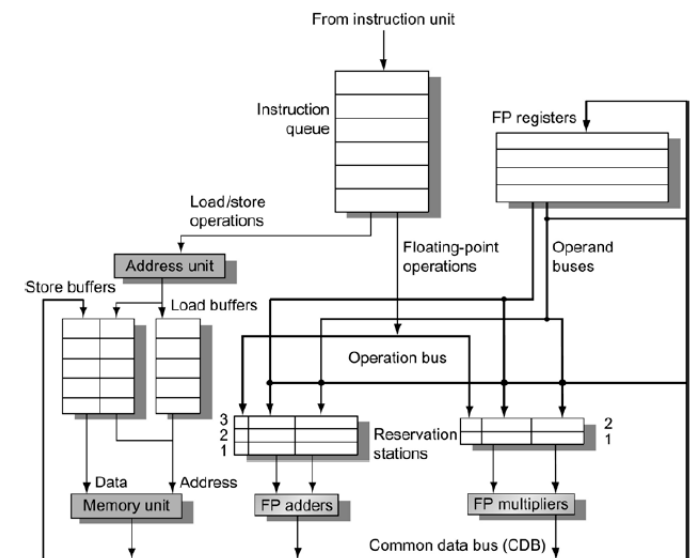
	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
i1: LDR r2, [r1,#0]	FE	DE	RN	RR	DI	IS	EX@	EXD\$...miss...			WB	RT			
i2: BNEZ r2, i7		FE	DE	RN	RR	DI	IS	IS	IS	IS	IS	IS	EX	WB	RT		
i3: ADD r2, r5, #2			FE	DE	RN	RR	DI	IS	EX	WB	RT	RT	RT	RT	RT		
i4: ADD r7, r2, #3				FE	DE	RN	RR	DI	IS	EX	WB	RT	RT	RT	RT		

In-order to Out-of-Order

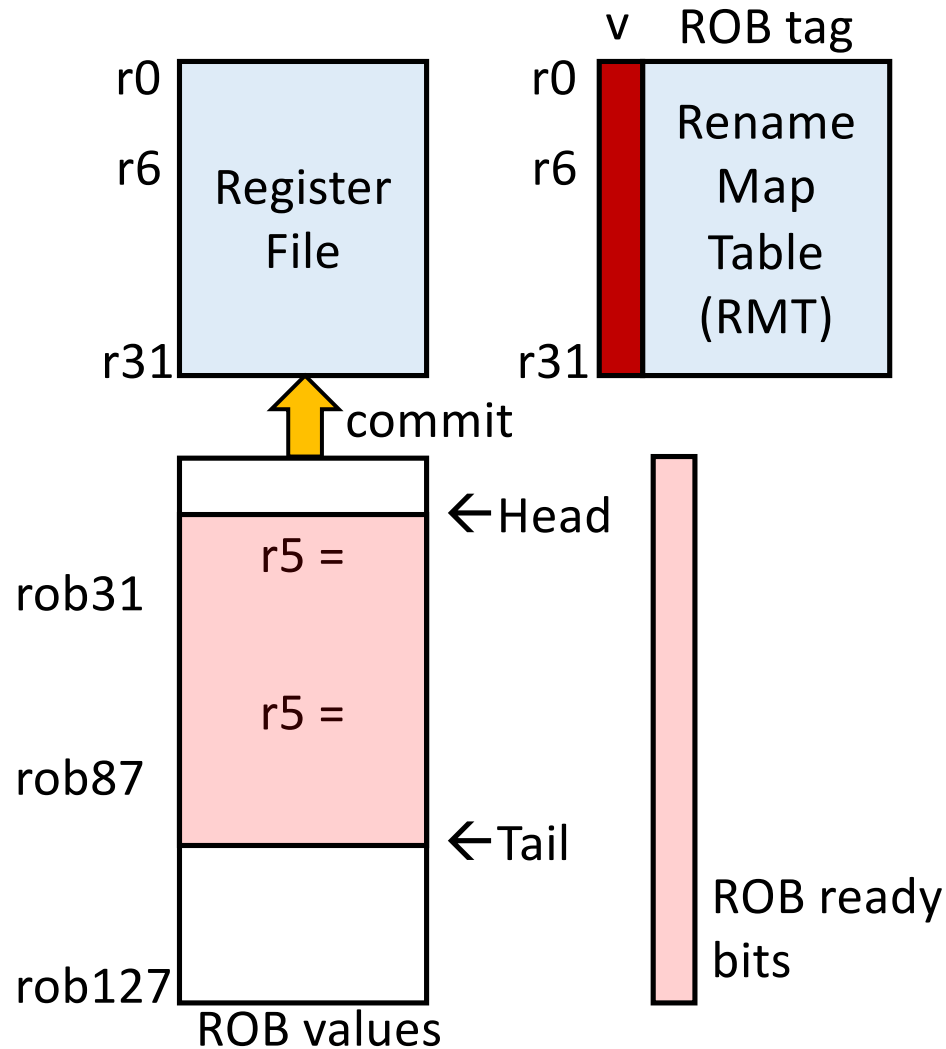


IBM 360/91 Floating Point Unit

- Due to Tomasulo's Algorithm [1967]
 - Execute multiple floating-point instructions concurrently
 - The original machine was imprecise (not a problem for floating point)
- Adding ROB is straightforward
- Stall on branch (limited ILP)



ARF + ROB Summary



- Physical register file = ARF + ROB
- Commit values by moving ROB value at head into ARF

Recovery

- Wait until exception/misprediction reaches head
- $T = H$
- Reset all "v" bits in RMT

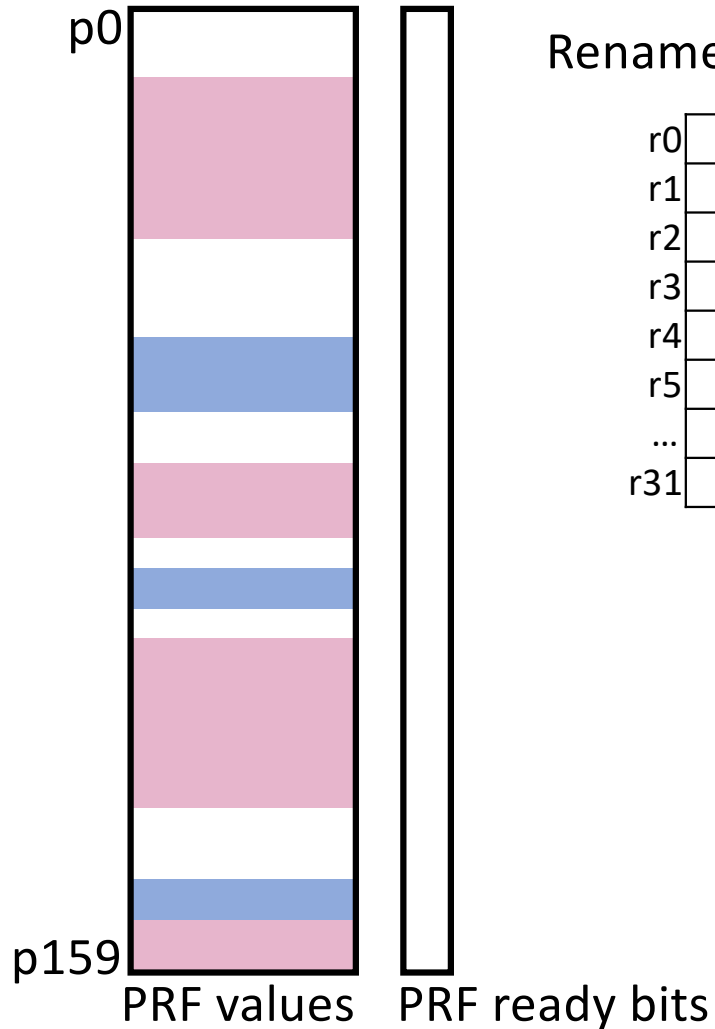
Revision: Main Concepts

- **Register renaming**
 - Rename logical registers to an extended set of physical registers
 - Avoid WAR and WAW hazards (main structure: ROB or RS/IQ)
- **Dynamic scheduling**
 - Send instructions to the functional units out of the original program order (IQ)
- **Speculation**
 - Predict branch outcomes and execute instructions before branches are resolved + have the ability to recover from mis-speculation (main structure: BPU/BTB/ROB)
- **Hardware speculation**
 - Dynamic branch prediction + dynamic scheduling + speculation
- **Precise interrupts**
 - On an exception, the architectural state must correspond to the sequential architectural model (main structure: ROB)

Drawbacks of ARF+ROB Design

- **Register Read stage before Issue stage**
 - Can't be after
 - If value is available at time of renaming, must grab it and “**capture**” it in the issue queue
 - Issue queue (IQ) needs to store values while waiting for all operands to be available
 - If IQ only kept pointer to value (ROB tag), value could move from ROB to ARF before instruction issues and then pointer is stale
- **Committing register values requires data movement**
 - Data movement (**ROB to ARF**) takes **extra cycles** and **consumes energy**

PRF Style



Rename Map Table

r0	p10	Phys. Reg. Tag
r1	p67	
r2	p11	
r3	p33	
r4	p46	
r5		
...		
r31	p2	

Compared to ARF + ROB

- A monolithic physical register file (PRF) provides an extended set of registers for renaming
- A subset of registers represent the architectural state
- RMT provides the mapping between architectural and physical registers
- *(pro) Committing & freeing registers does not require data movement*
- *(con) Restoring RMT is not a simple flash-clear of bits (still conceptually similar)*

Intel Sandy Bridge

<https://www.anandtech.com/show/3922/intels-sandy-bridge-architecture-exposed/3>

A Physical Register File (Copying from the link here for your benefit)

Just like AMD announced in its [Bobcat and Bulldozer architectures](#), in Sandy Bridge Intel moves to a physical register file. In Core 2 and Nehalem, every micro-op had a copy of every operand that it needed. This meant the out-of-order execution hardware (scheduler/reorder buffer/associated queues) had to be much larger as it needed to accommodate the micro-ops as well as their associated data. Back in the Core Duo days that was 80-bits of data. When Intel implemented SSE, the burden grew to 128-bits. With AVX however we now have potentially 256-bit operands associated with each instruction, and the amount that the scheduling/reordering hardware would have to grow to support the AVX execution hardware Intel wanted to enable was too much.

A physical register file stores micro-op operands in the register file; **as the micro-op travels down the OoO engine it only carries pointers to its operands and not the data itself**. This significantly reduces the power of the out of order execution hardware (moving large amounts of data around a chip eats tons of power), it also reduces die area further down the pipe. The die savings are translated into a larger out of order window.

The die area savings are key as they enable one of Sandy Bridge's major innovations: AVX performance.

Loads and Stores

- Loads and stores also execute out of order
- Store **Cancels** all speculative (younger) loads with matching addresses
- Load searches for all speculative (older) stores with matching addresses
 - It gets the best it can (cache, main memory, ROB, RF)
- Once we have speculation support, we can predict other things
 - Speculating on register values (value prediction)!