COMP2300-COMP6300-ENGN2219
Computer Architecture

Convener: 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 ILP
  - Scalar in-order pipeline (partial overlap)
  - Superscalar in-order pipeline (full overlap)

- Full overlap requires multiple functional units and datapaths

- ILP processors bet that many instructions in the program are independent of each other

- We **will now see more aggressive ILP exploitation**

Dependences and Hazards

- Uncovering ILP aggressively requires understanding dependences and hazards

- **Recall:** Dependence is a program's property

- **Recall:** Hazard is a microarchitecture property

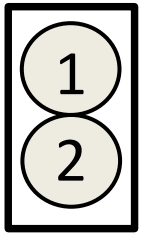
  - We have seen
    - Read-after-write hazard (due to true dependences)
    - Control hazard (due to branches in programs)

True dependence

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

- Data or true dependence
  - **2** needs the result of **1**

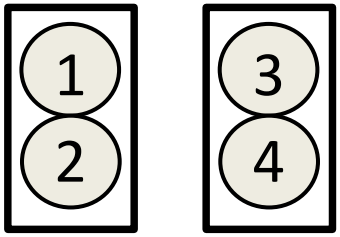
- We have a single (dependent) instruction chain



True dependences

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

- Two independent instruction chains



Anti dependence

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

- Anti dependence
  - 2 needs to write what 1 needs to read

- Anti dependence is an example of false dependence
  - One way to eliminate this dependence is to replace R6 in instruction # 2 with a different register

Output dependence

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

- Output dependence
  - 1 and 2 wants to write to the same register

- Again, it is a false dependence
  - We can replace register R2 in instruction # 2 with a different register to eliminate the dependence

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**
  - None of the dependences result in a hazard (no concurrent execution)
- **Basic ARM 5-stage in-order pipeline**
  - Multiple instructions in different stages (possibility of RAW)

- OOO CPUs we will study exhibit WAR and WAW hazards

From In-Order to Out-of-Order

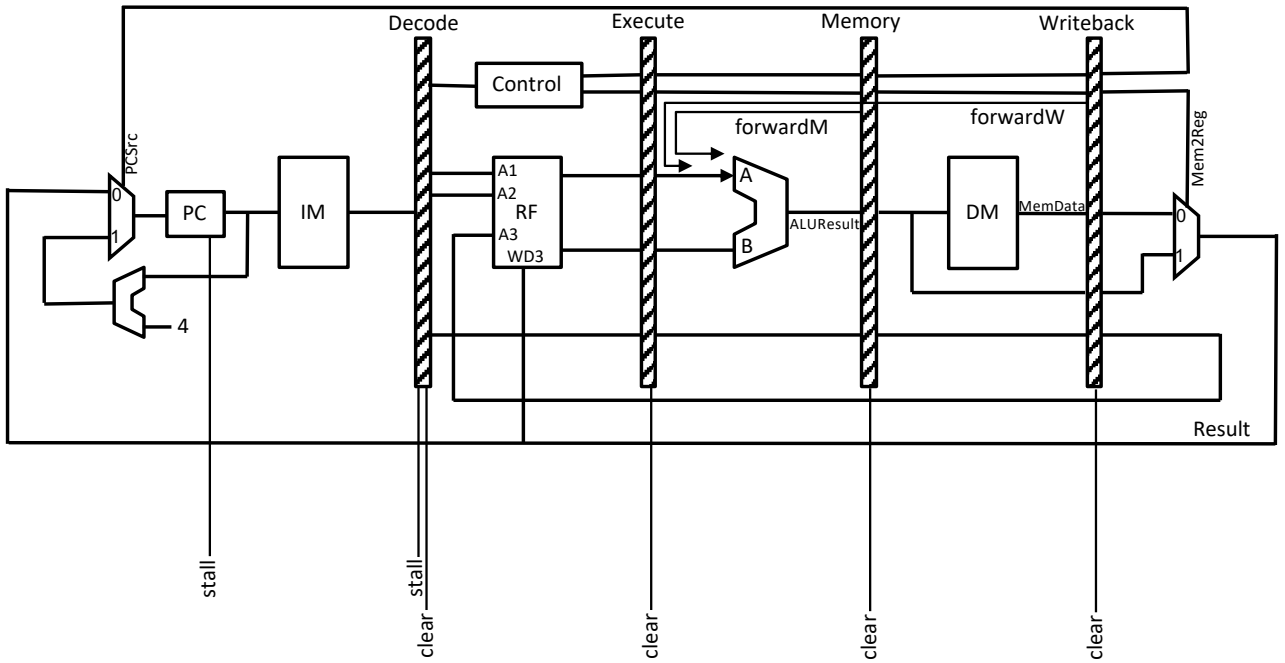
In-Order Pipeline

- **In-order pipeline:** Instructions are fetched, decoded, and executed in program order
- CPU hardware does no **reordering** of instructions to avoid stalls or execute independent instructions during a cache miss
- The simple **in-order** pipeline we have seen exploits **limited** parallelism
- It "completely" stalls because of
  - Cache misses
  - Floating point multiply and divide (long-latency operations)
- We will make it more **aggressive** and see that there is still a problem

In-Order Pipeline w/t Cache

- Add a data cache and realistic data memory with 100 cycle latency
- **Stalls on a cache miss due to structural dependency**
  - One ALU capable of executing "one operation" at a time
  - Memory can only handle "one request" at a time
  - If the **MEM** stage stalls, then EX stalls, and the entire pipeline stalls (aka. stall-on-miss pipeline)

In-Order Pipeline w/t Cache

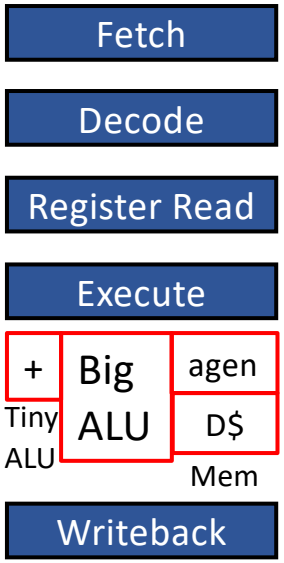


Aggressive In-Order Pipeline/Cache

- We will use a different more **"aggressive"** in-order pipeline to understand problem with in-order pipelines
  - Many ALUs and memory capable of handling multiple requests at a time
  - Eliminating the **structural** dependency and rearranging stages
  - Fetches, decodes, and issues one instruction **in order** every cycle to the functional units, but many of them can be in execution in any cycle

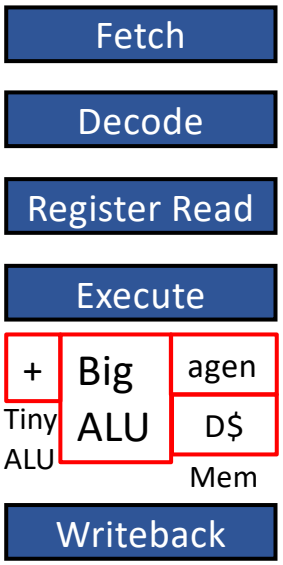
Stall-on-Use Pipeline

- Decode and register read stages are separate
- Memory access happens in the execute stage
- Memory can handle multiple memory requests
- A data cache is used to exploit locality
- Register read stage also implements the issue policy
  - Issuing is sending an instruction to the ALU for execution when operands are ready



Stall-on-Use Pipeline

- Assumptions about **execute** stage
  - Non-blocking execute stage (multiple functional units)
  - **Many** functional units (concurrent instruction execution)
  - **Instructions do not stall due to structural dependency**
  - Load is divide into:
    - address generation (agen)
    - data cache access (D$)
    - memory access (if load miss or cache miss)



- Assumptions about issue logic in RR stage:
  - **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

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												
i2													
i3													
i4													

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

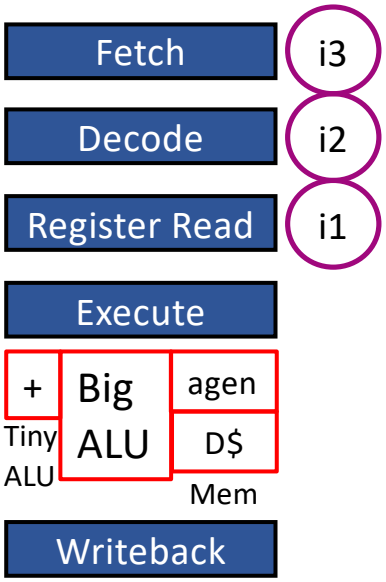
i1: LDR R2, [R1, #0]

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											
i2		FE											
i3													
i4													



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										
i2		FE	DE										
i3			FE										
i4													

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	$EX_{@}$									
i2		FE	DE	RR									
i3			FE	DE									
i4				FE									

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	$EX_{@}$	$EX_{D\$}$	...miss...							
i2		FE	DE	RR	EX	WB							
i3			FE	DE	RR	EX							
i4				FE	DE	RR							

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

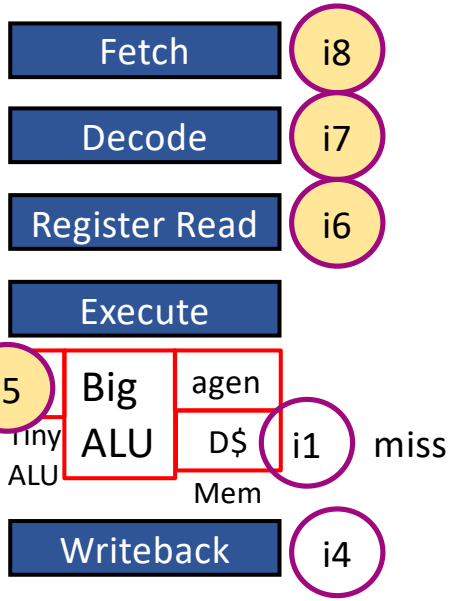
i1: LDR R2, [R1, #0]

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					

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					

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

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

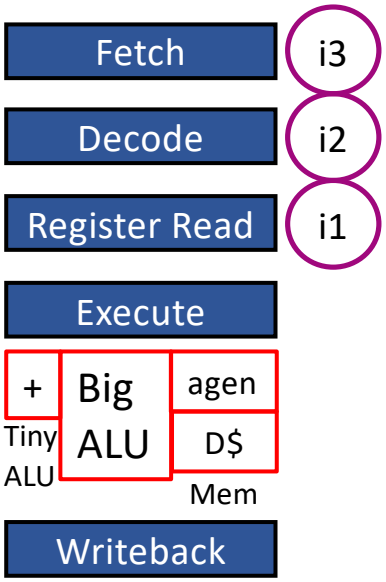
```
i1: LDR  R2,  [R1,  #0]
i2: ADD  R4,  R2,  #1
i3: ADD  R6,  R5,  #2
i4: ADD  R7,  R6,  #3
```

	1	2	3	4	5	6	7	8	9	10	11	12	13
i1	FE												
i2													
i3													
i4													

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											
i2		FE											
i3													
i4													



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

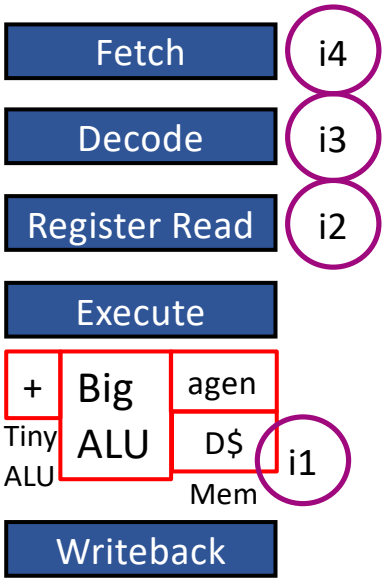
```
i1: LDR  R2,  [R1,  #0]
i2: ADD  R4,  R2,  #1
i3: ADD  R6,  R5,  #2
i4: ADD  R7,  R6,  #3
```

	1	2	3	4	5	6	7	8	9	10	11	12	13
i1	FE	DE	RR										
i2		FE	DE										
i3			FE										
i4													

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
```

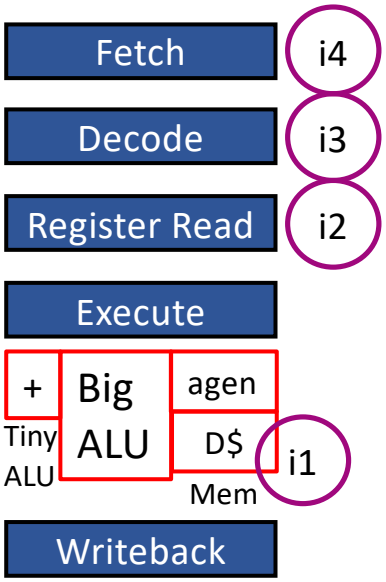
	1	2	3	4	5	6	7	8	9	10	11	12	13
i1	FE	DE	RR	$EX_{@}$									
i2		FE	DE	RR									
i3			FE	DE									
i4				FE									



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
```

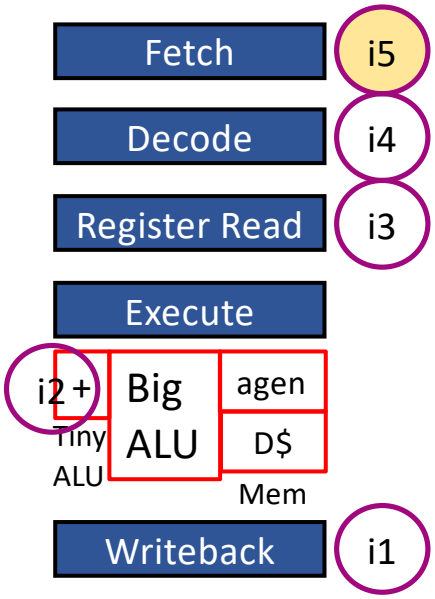
	1	2	3	4	5	6	7	8	9	10	11	12	13
i1	FE	DE	RR	$EX_{@}$	$EX_{D\$}$	...miss...							
i2		FE	DE	RR	RR								
i3			FE	DE	DE								
i4				FE	FE								



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
```

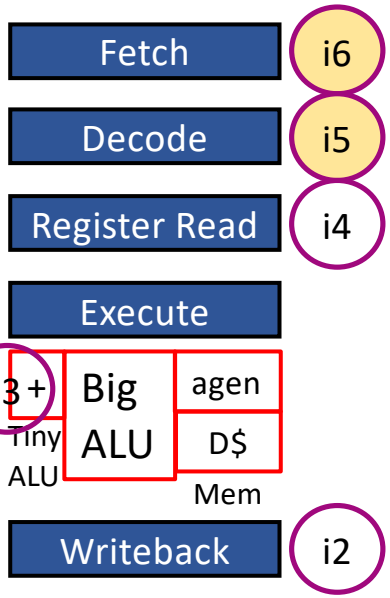
	1	2	3	4	5	6	7	8	9	10	11	12	13
i1	FE	DE	RR	$EX_{@}$	$EX_{D\$}$	...miss...							
i2		FE	DE	RR	RR	RR	RR	RR	RR				
i3			FE	DE	DE	DE	DE	DE	DE				
i4				FE	FE	FE	FE	FE	FE				



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
```

	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
```

	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, hindering ILP exploitation

What have we established?

- `i2` must wait for `i1`
  - `i2` depends on `i1`
- `i3`, `i4` do not need to wait for the `i1`–`i2` dependent chain (they are independent)
- But the `i3`–`i4` chain stalls
  - **Key insight:** In-order issue translates into a structural hazard
  - RR stage (issue stage) blocked by the stalled `i2`

- **In-order issue policy is the problem**
  - 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!

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 (OOO)

Issue Queue

- OOO pipeline unblocks RR (issue) using a new **instruction buffer** for stalled data-dependent instructions
  - It is called **“reservation stations”**, **“issue buffer”**, **“issue queue”**, **“scheduler”**, **“scheduling window”**
- 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 better

Dynamic Scheduling

Instruction Scheduling

- **Deciding the order in which instructions are executed, where each ordering is called a schedule**
  - The goal is to maximize ILP
  - All dependences must be respected

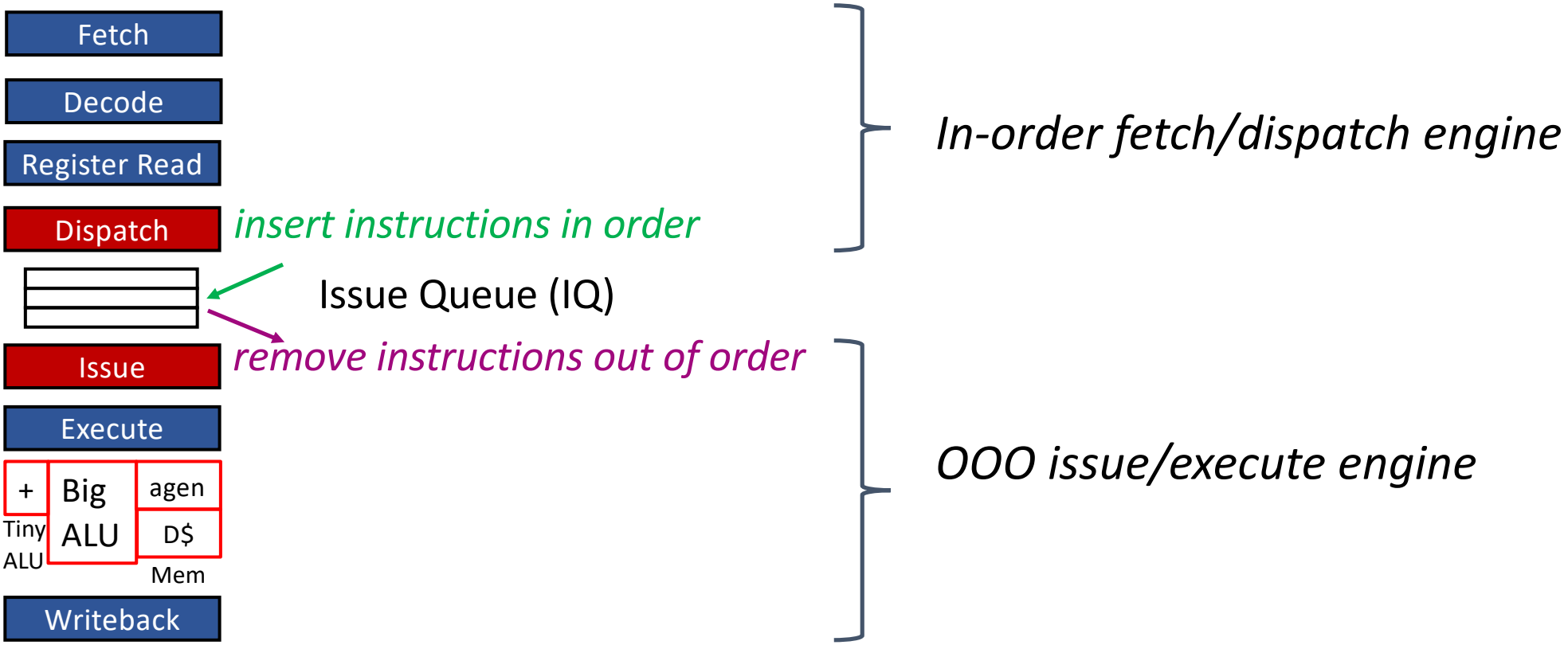
- **Static scheduling**
  - Compiler decides the ordering of instruction execution
  - It relies on hardware for correct execution (hazard detection)
  - ARM 5-stage pipeline is *statically scheduled*
  - 2-way in-order superscalar we saw before is also *statically scheduled*

- **Dynamic scheduling**
  - Hardware decides the ordering transparently to software
  - "Smart hardware, dumb software"

What is dynamic scheduling?

- Deciding which instructions to execute next, ***using extra hardware (i.e., stalled instructions do not impede instruction fetch)***
- In a dynamically scheduled pipeline, younger, ready instructions issue and execute out of order w.r.t. older non-ready instructions
- Issue queue opens up the pipeline to future independent instructions
- Dynamically scheduled processors are superscalar, but we will assume scalar execution to simplify illustration of principles

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



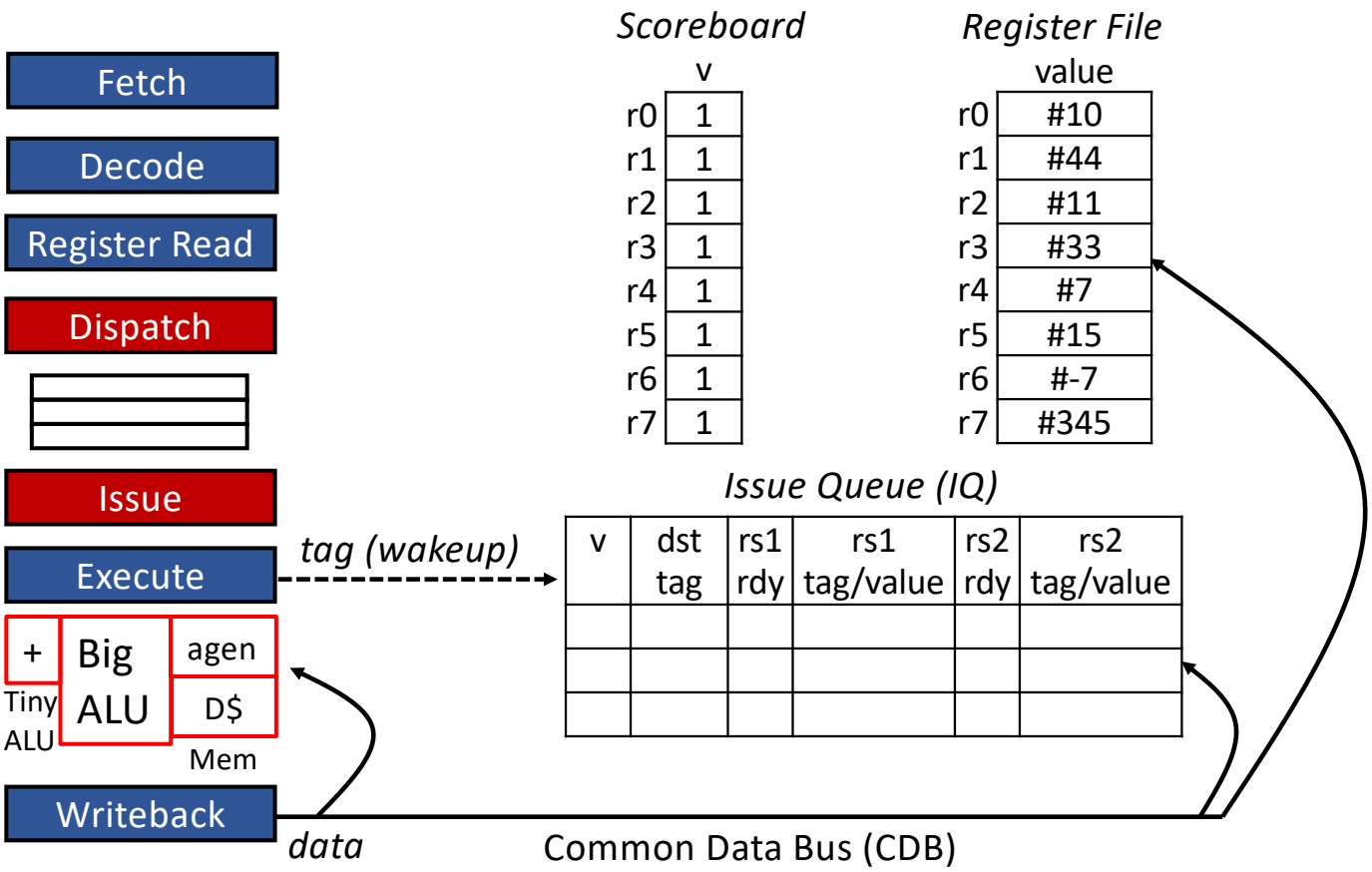
Version # 1: CDC 6600 Scoreboard

- The idea of a scoreboard was introduced by Control Data Corporation (CDC) in **CDC6600** machine

- **New stages**
  - Dispath stage
  - Issue stage

- **New components**
  - Scoreboard

- Common Data Bus (CDB) for data writeback and forwarding


Fetch
Decode
Register Read
Dispatch
Issue
Execute
+
Big
ALU
agen
D$
Tiny
ALU
Mem
Writeback
Scoreboard
v
r0 1
r1 1
r2 1
r3 1
r4 1
r5 1
r6 1
r7 1
Register File
value
r0 #10
r1 #44
r2 #11
r3 #33
r4 #7
r5 #15
r6 #-7
r7 #345
Issue Queue (IQ)
v
dst tag
rs1 rdy
rs1 tag/value
rs2 rdy
rs2 tag/value
tag (wakeup)
data
Common Data Bus (CDB)

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

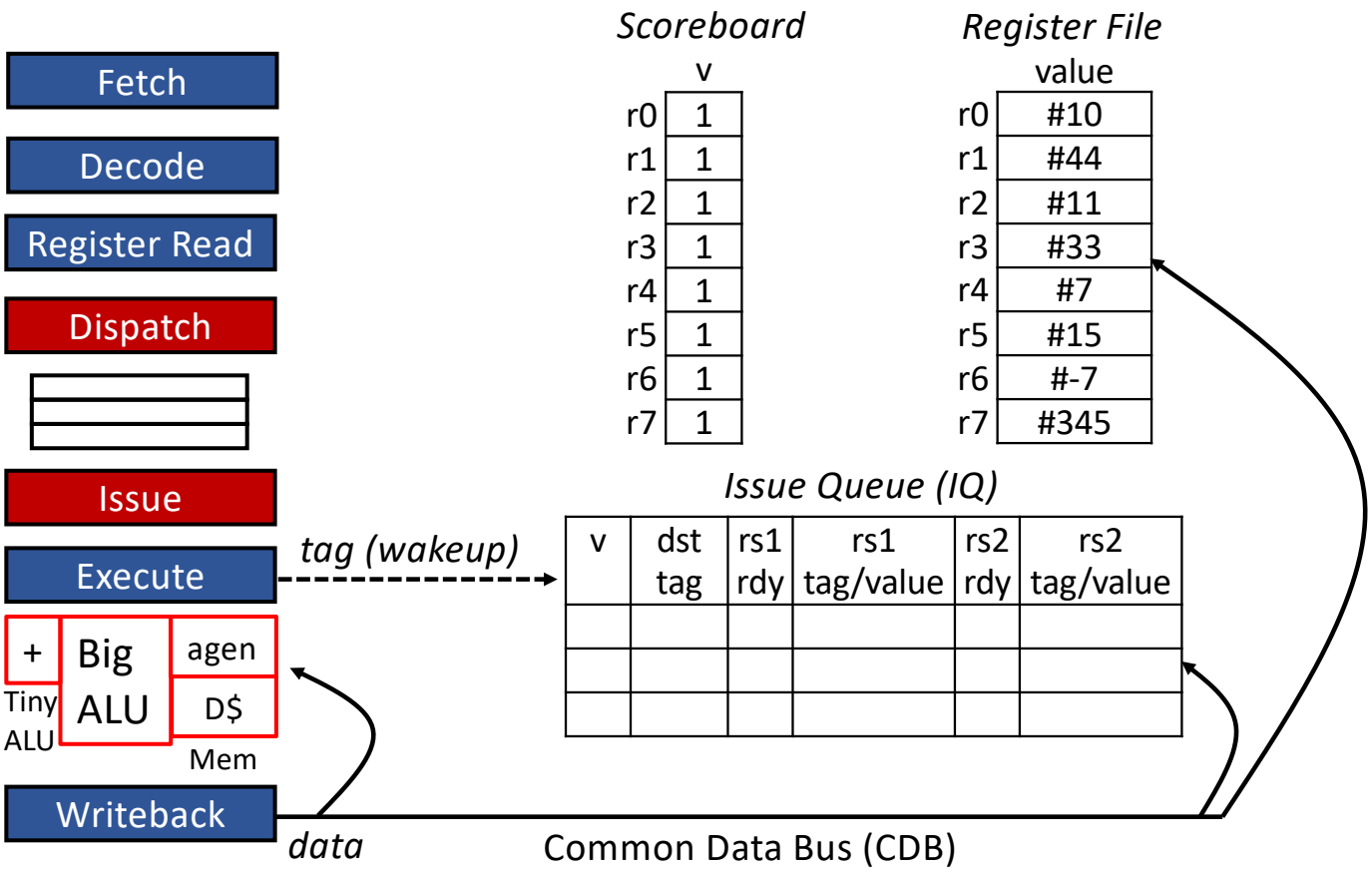
- **Issue stage**
  - If both operands are ready, the issue stage sends the instruction to the execution unit
  - Deallocate the issue queue entry by setting **v** = 0

- **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)

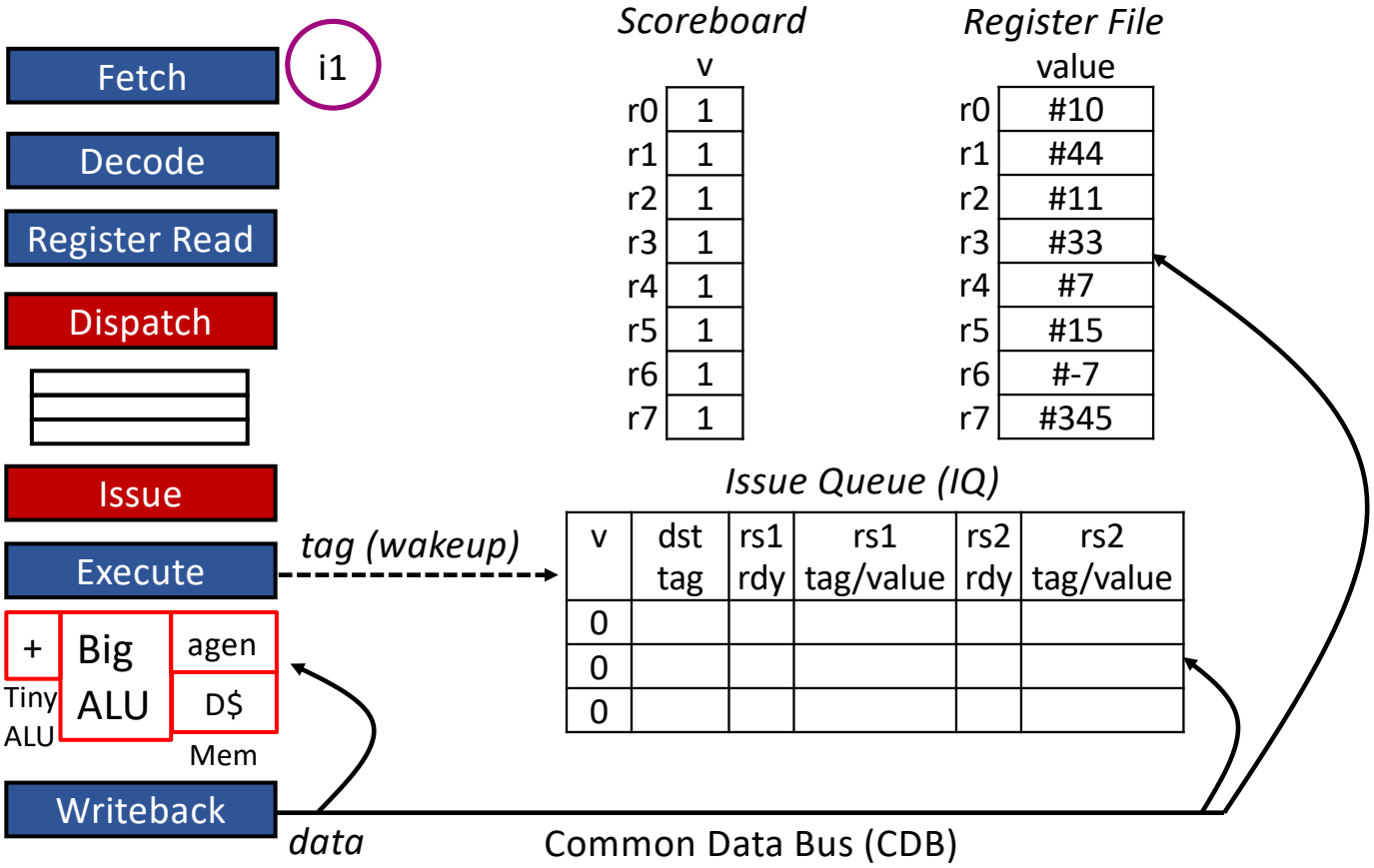
CDC 6600 Scoreboard (Key Additions)

- **How are instructions selected for execution?**
  - The issue queue waits for destination tags to appear
  - When the destination tag appears, it **wakes up** all instructions waiting for that tag
  - One of the "ready" instructions is sent to the execute next
  - **Easy way to remember:** Capture-Tag-And-Go-For-Execution

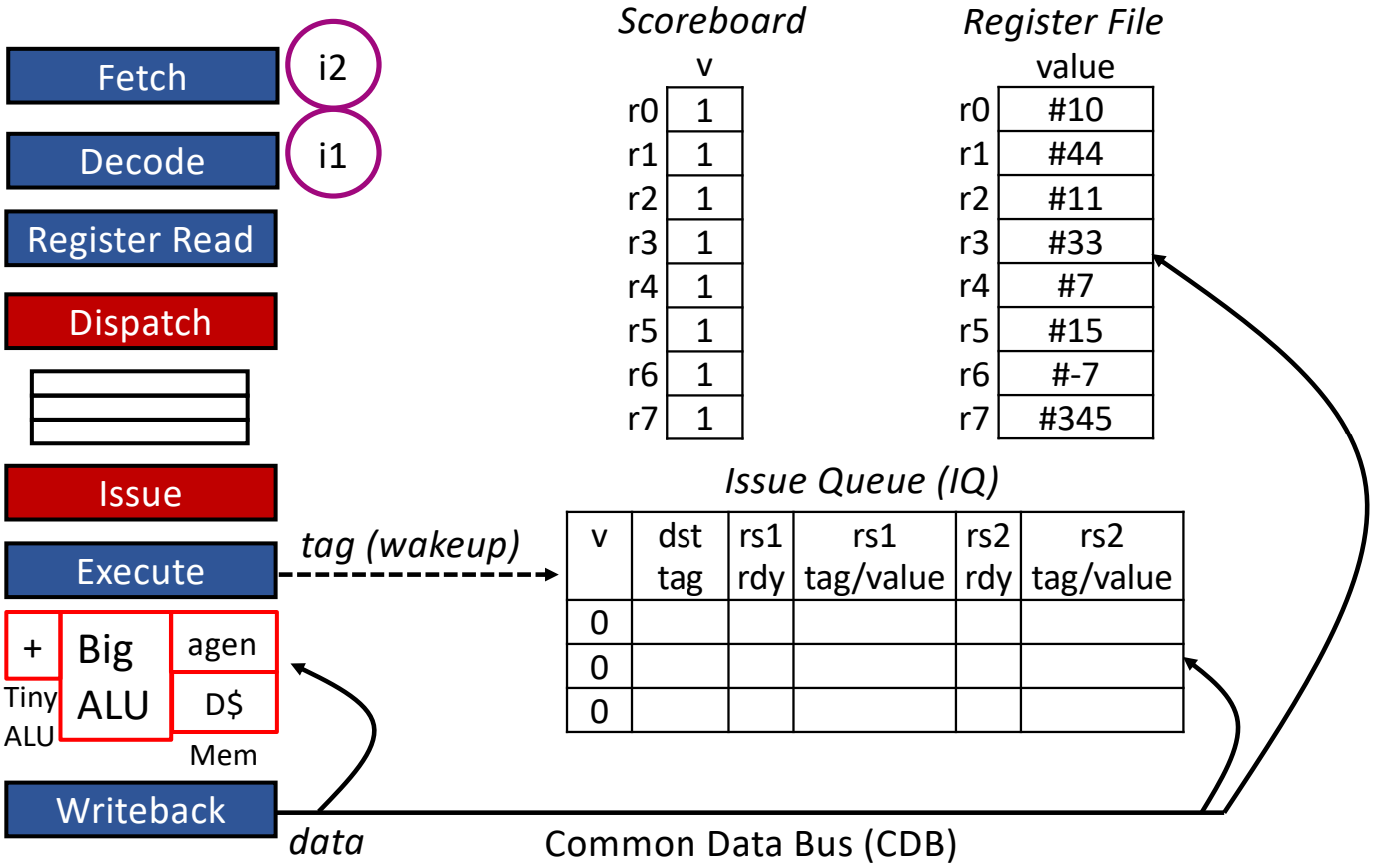
- **Common Data Bus (CDB)**
  - The values are broadcasted over the CDB bypassing register file writes
  - This bus implements forwarding
  - Values are forwarded to the issue queue, register file, and execute stage



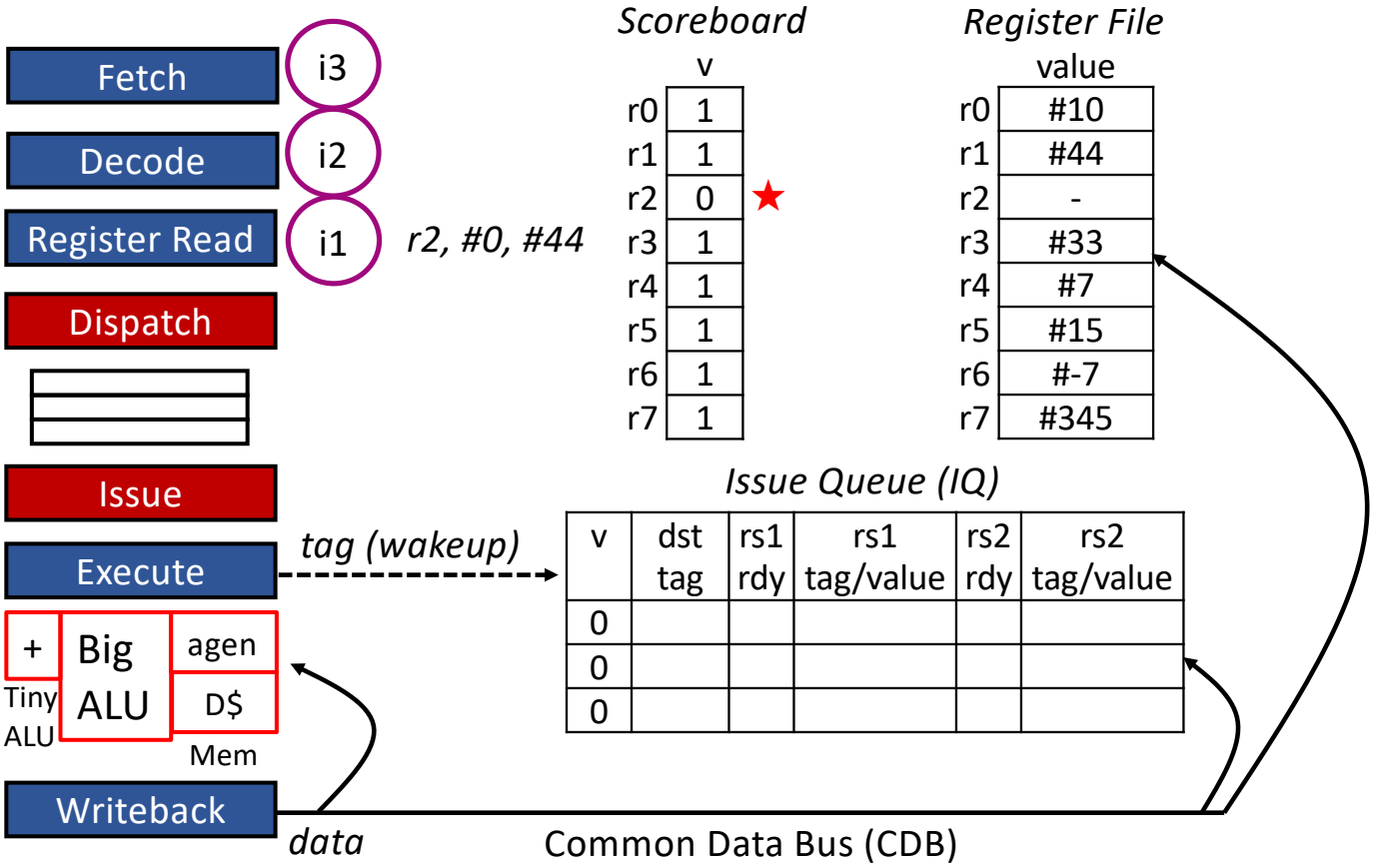
	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
i1: LDR r2, [r1,#0]																	
i2: ADD r4, r2, #1																	
i3: ADD r6, r5, #2																	
i4: ADD r7, r6, #3																	



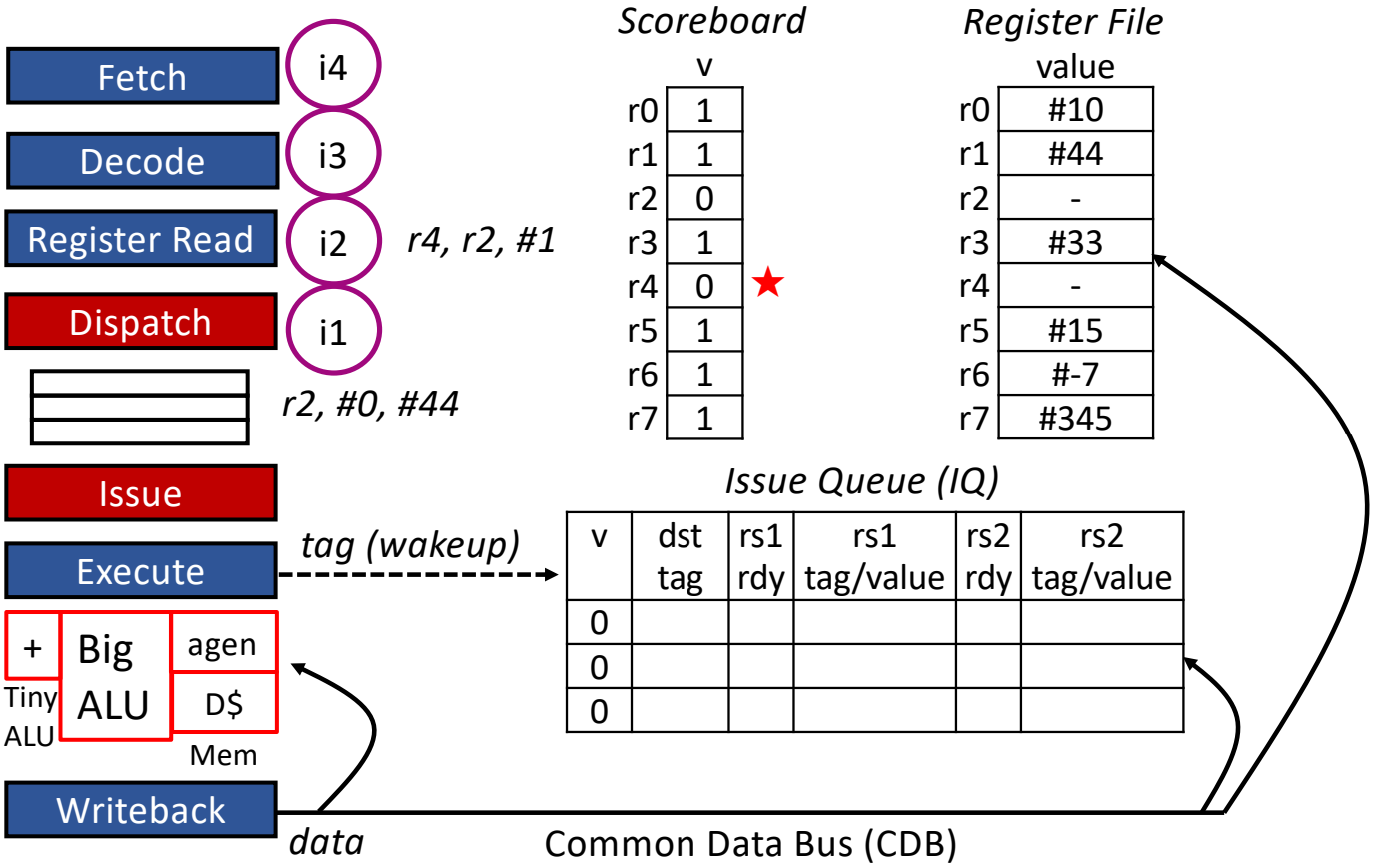
	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
i1: LDR r2, [r1,#0]	FE																
i2: ADD r4, r2, #1																	
i3: ADD r6, r5, #2																	
i4: ADD r7, r6, #3																	



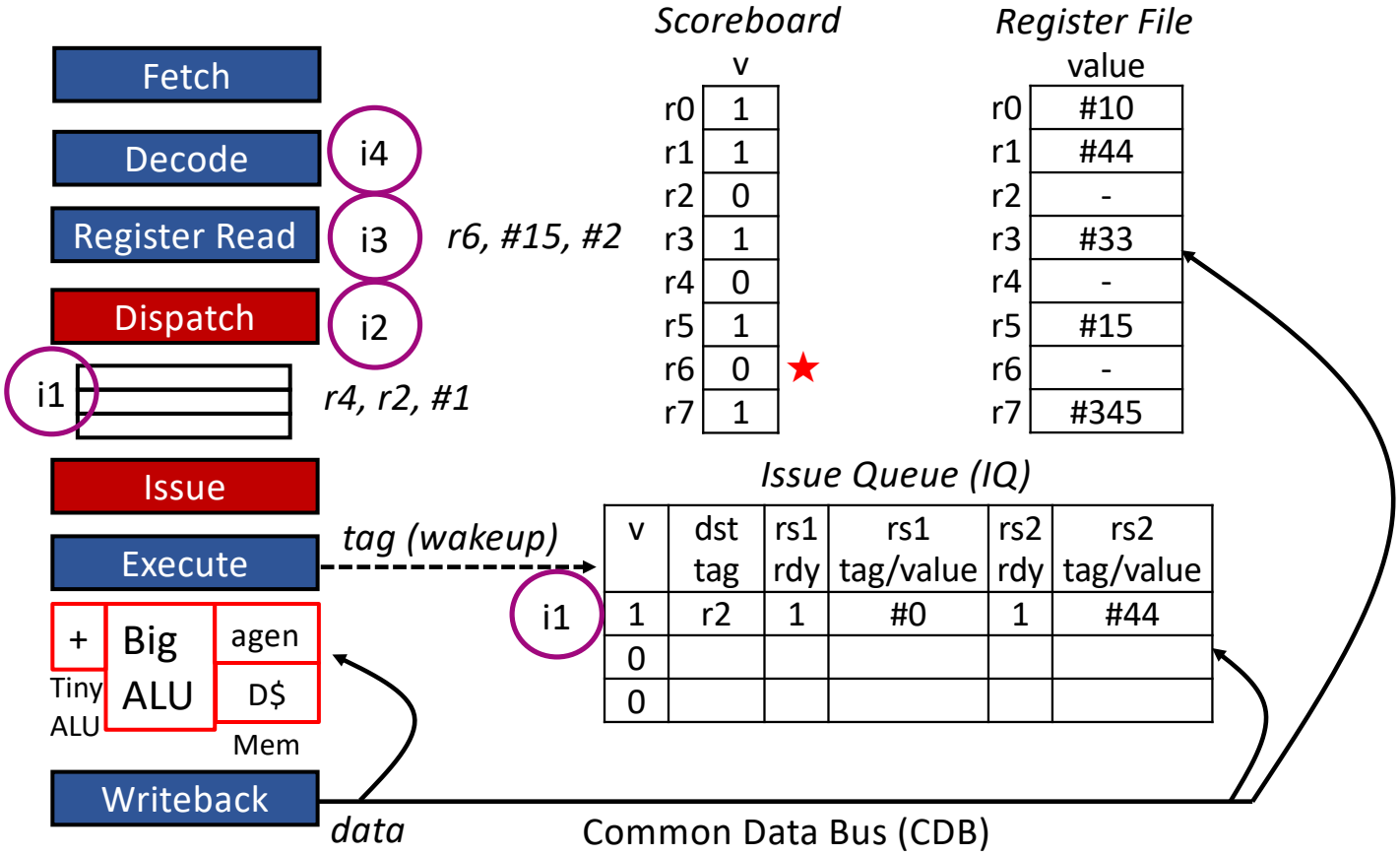
	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
i1: LDR r2, [r1,#0]	FE	DE															
i2: ADD r4, r2, #1		FE															
i3: ADD r6, r5, #2																	
i4: ADD r7, r6, #3																	



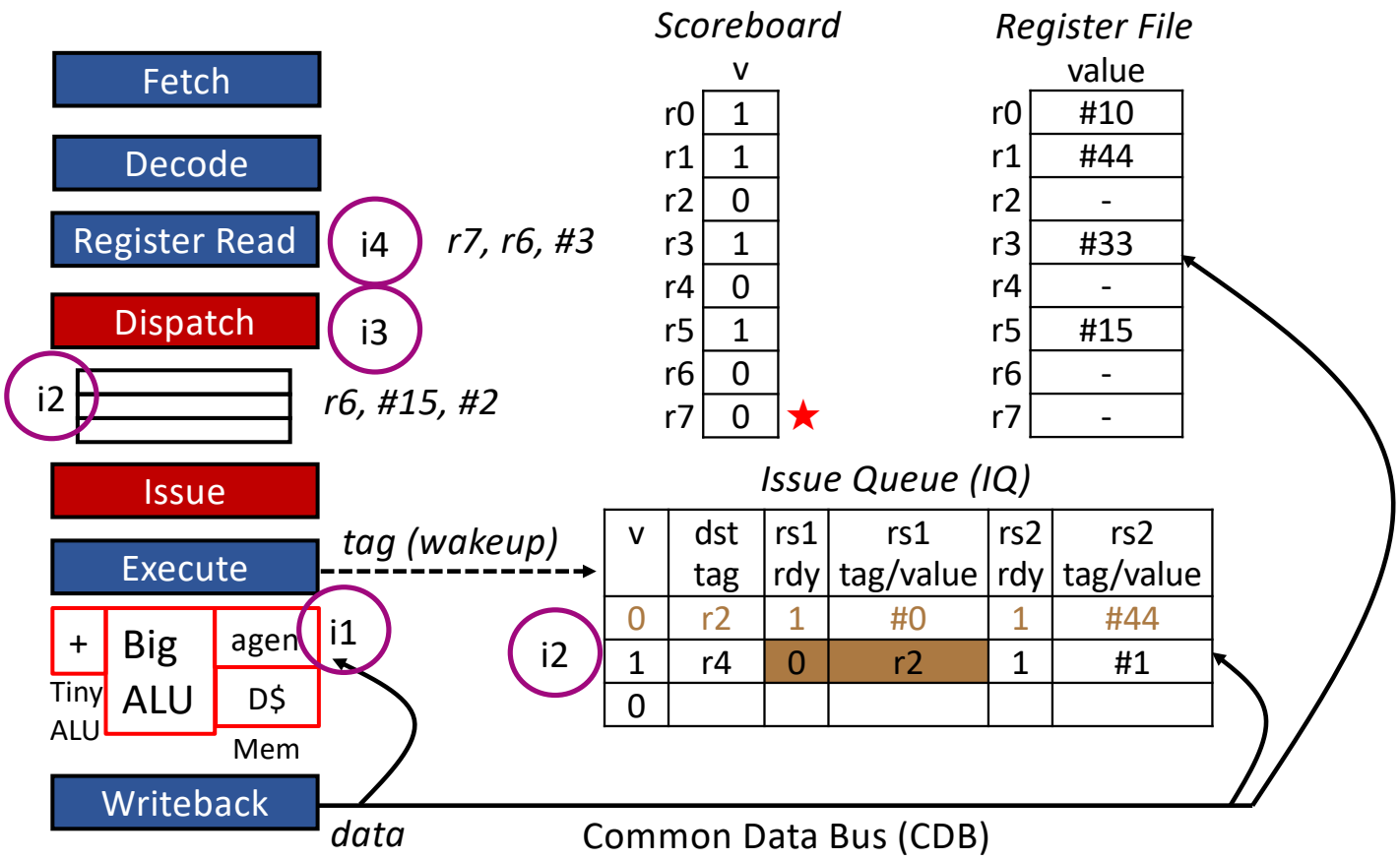
	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
i1: LDR r2, [r1,#0]	FE	DE	RR														
i2: ADD r4, r2, #1		FE	DE														
i3: ADD r6, r5, #2			FE														
i4: ADD r7, r6, #3																	



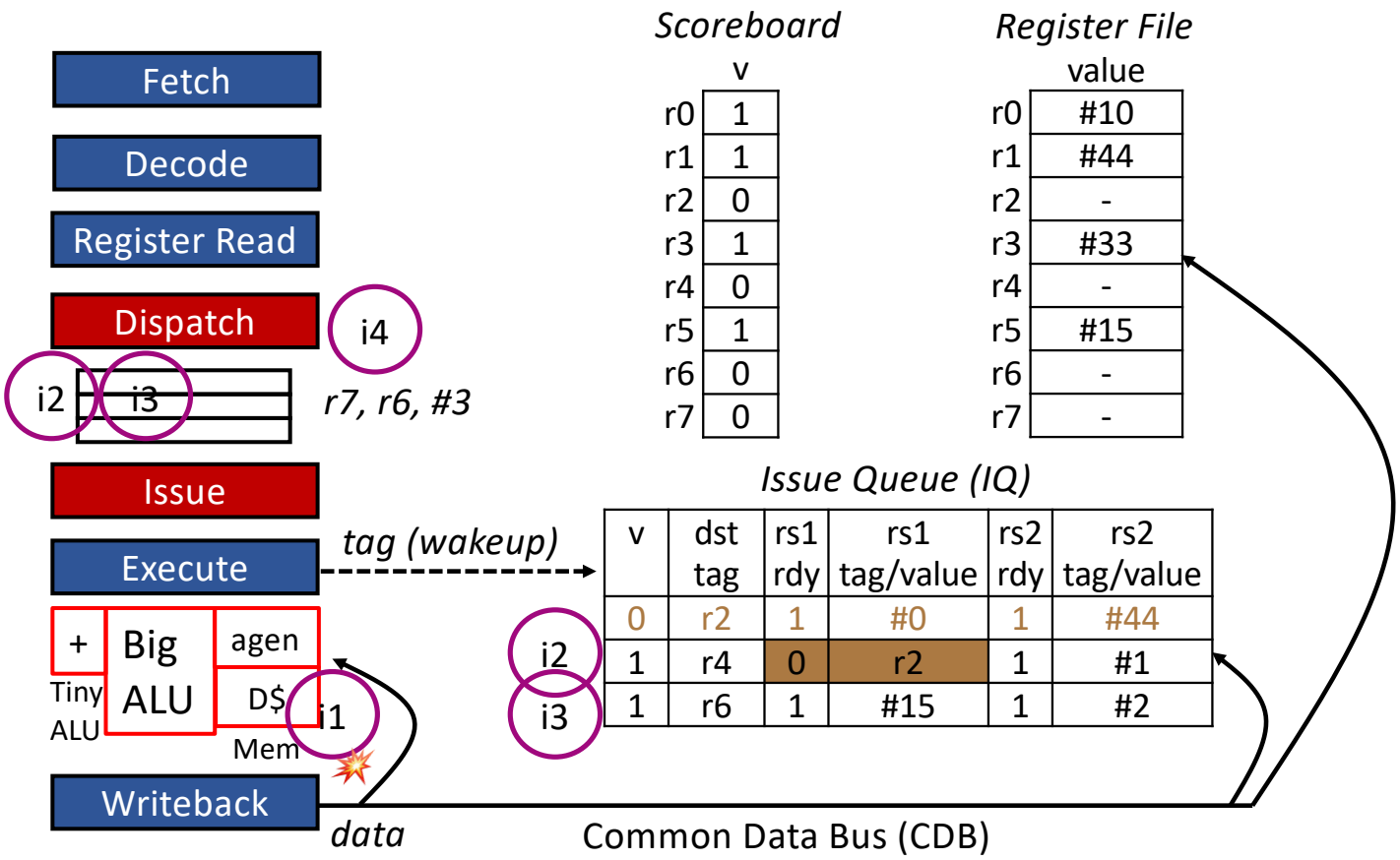
	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
i1: LDR r2, [r1,#0]	FE	DE	RR	DI													
i2: ADD r4, r2, #1		FE	DE	RR													
i3: ADD r6, r5, #2			FE	DE													
i4: ADD r7, r6, #3				FE													



	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												
i2: ADD r4, r2, #1		FE	DE	RR	DI												
i3: ADD r6, r5, #2			FE	DE	RR												
i4: ADD r7, r6, #3				FE	DE												

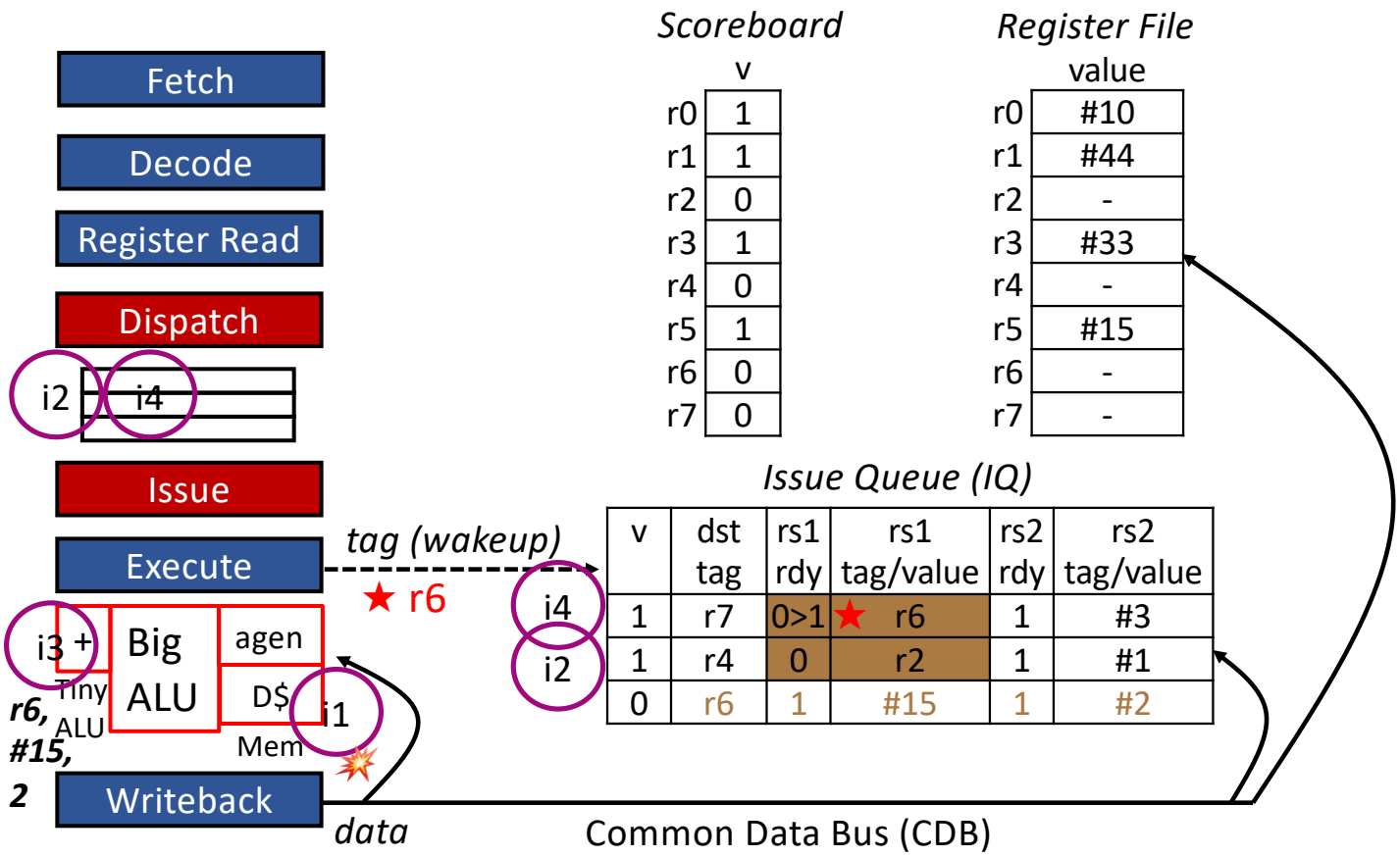


	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
i1: LDR r2, [r1,#0]	FE	DE	RR	DI	IS	$EX_{@}$											
i2: ADD r4, r2, #1		FE	DE	RR	DI	IS											
i3: ADD r6, r5, #2			FE	DE	RR	DI											
i4: ADD r7, r6, #3				FE	DE	RR											



cache miss r2, @44

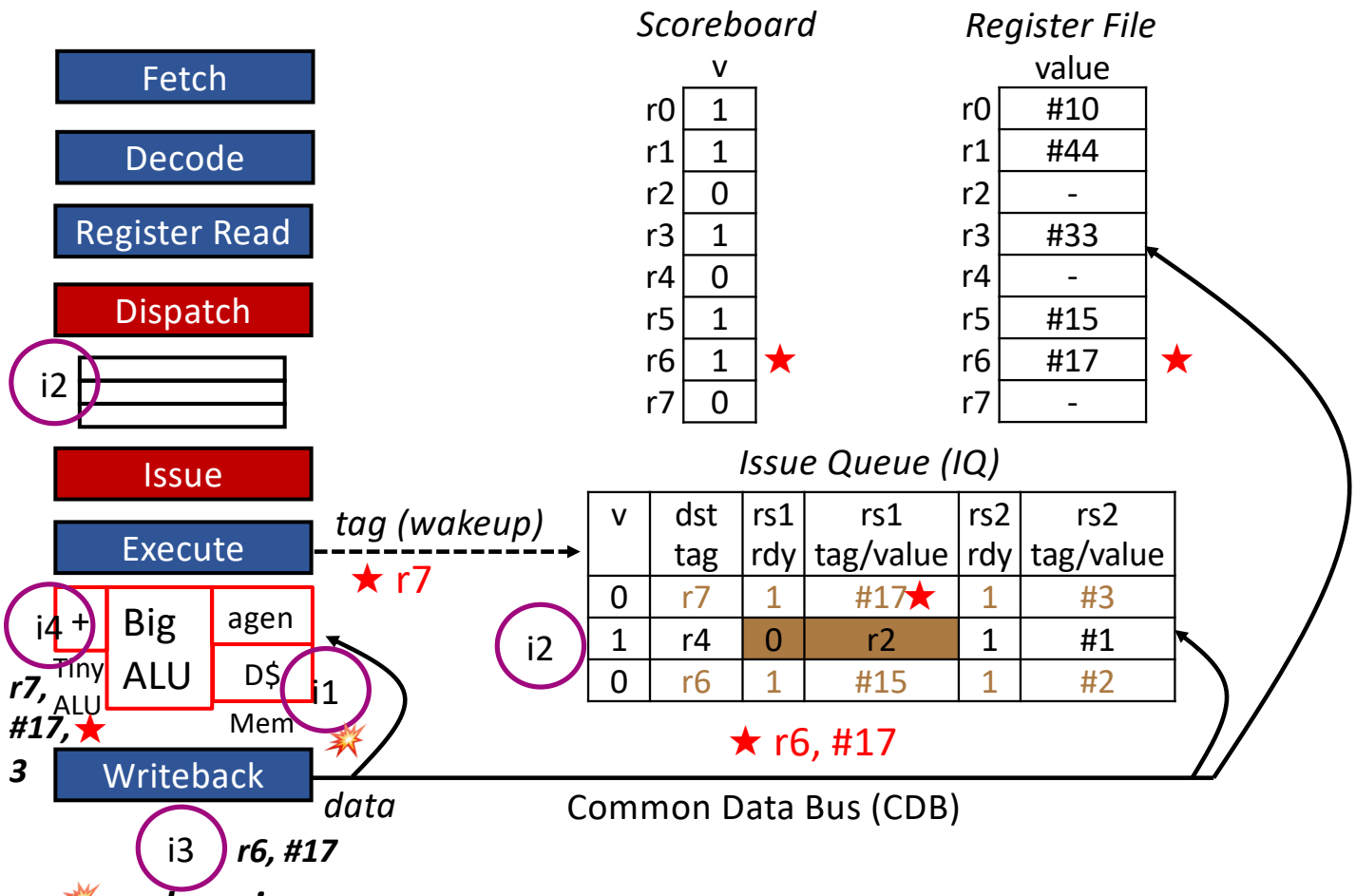
	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
i1: LDR r2, [r1,#0]	FE	DE	RR	DI	IS	$EX_{@}$	$EX_{D\$}$	… miss …									
i2: ADD r4, r2, #1		FE	DE	RR	DI	IS	IS										
i3: ADD r6, r5, #2			FE	DE	RR	DI	IS										
i4: ADD r7, r6, #3				FE	DE	RR	DI										



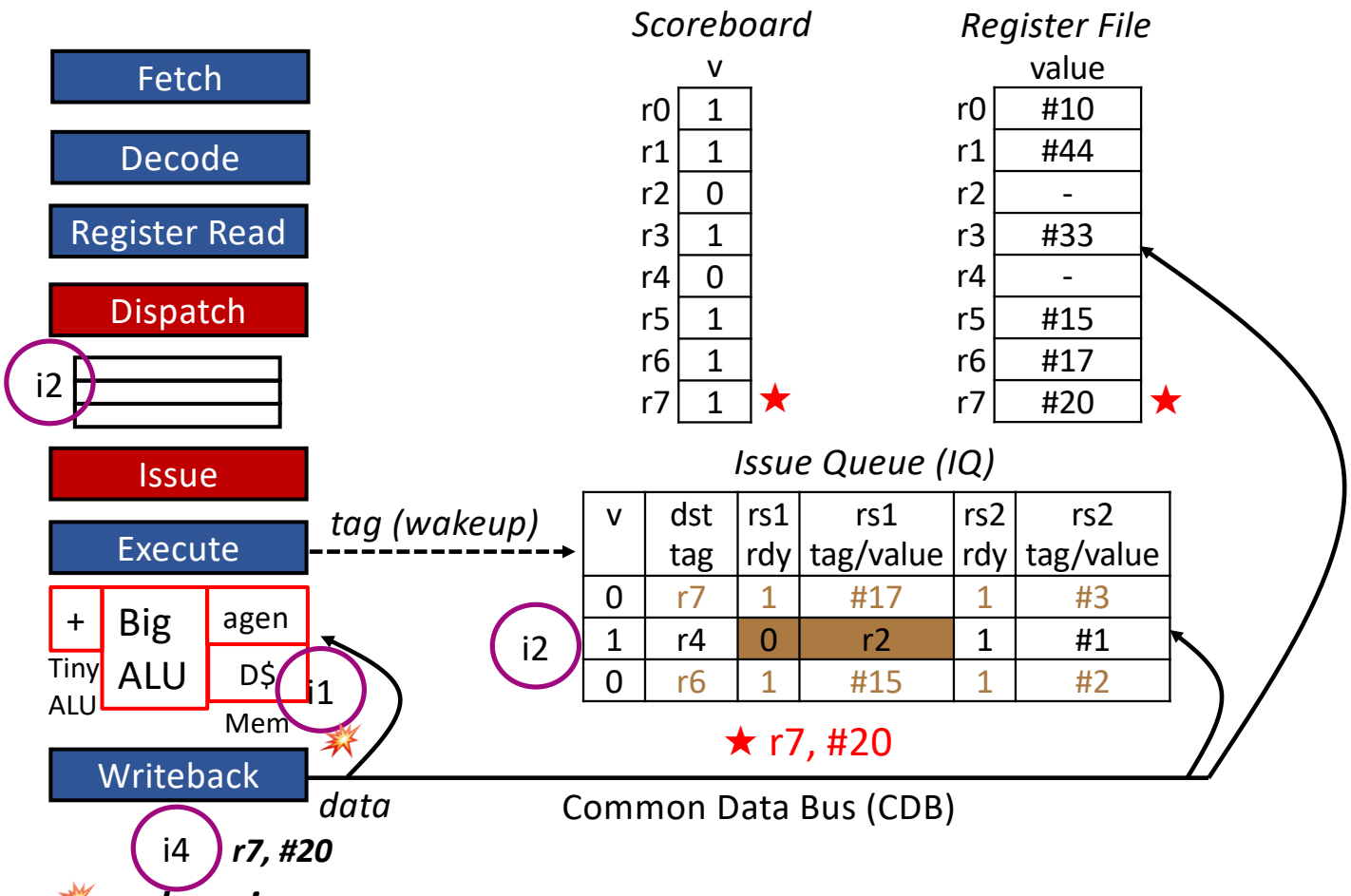
✸ ***cache miss***

r2, @44

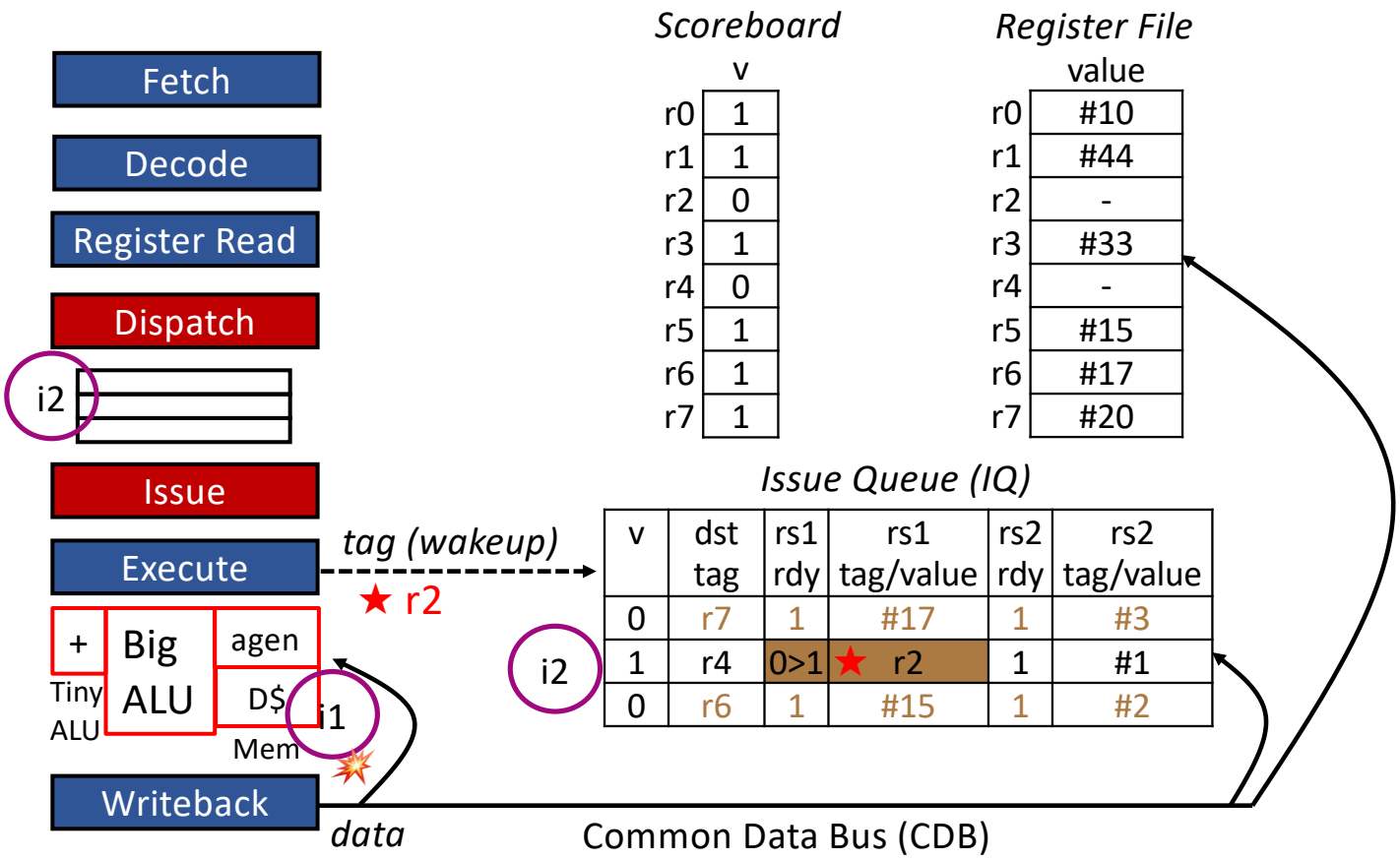
	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
i1: LDR r2, [r1,#0]	FE	DE	RR	DI	IS	$EX_{@}$	$EX_{D\$}$		… miss …								
i2: ADD r4, r2, #1		FE	DE	RR	DI	IS	IS	IS									
i3: ADD r6, r5, #2			FE	DE	RR	DI	IS	EX									
i4: ADD r7, r6, #3				FE	DE	RR	DI	IS									



	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								
i3: ADD r6, r5, #2			FE	DE	RR	DI	IS	EX	WB								
i4: ADD r7, r6, #3				FE	DE	RR	DI	IS	EX								



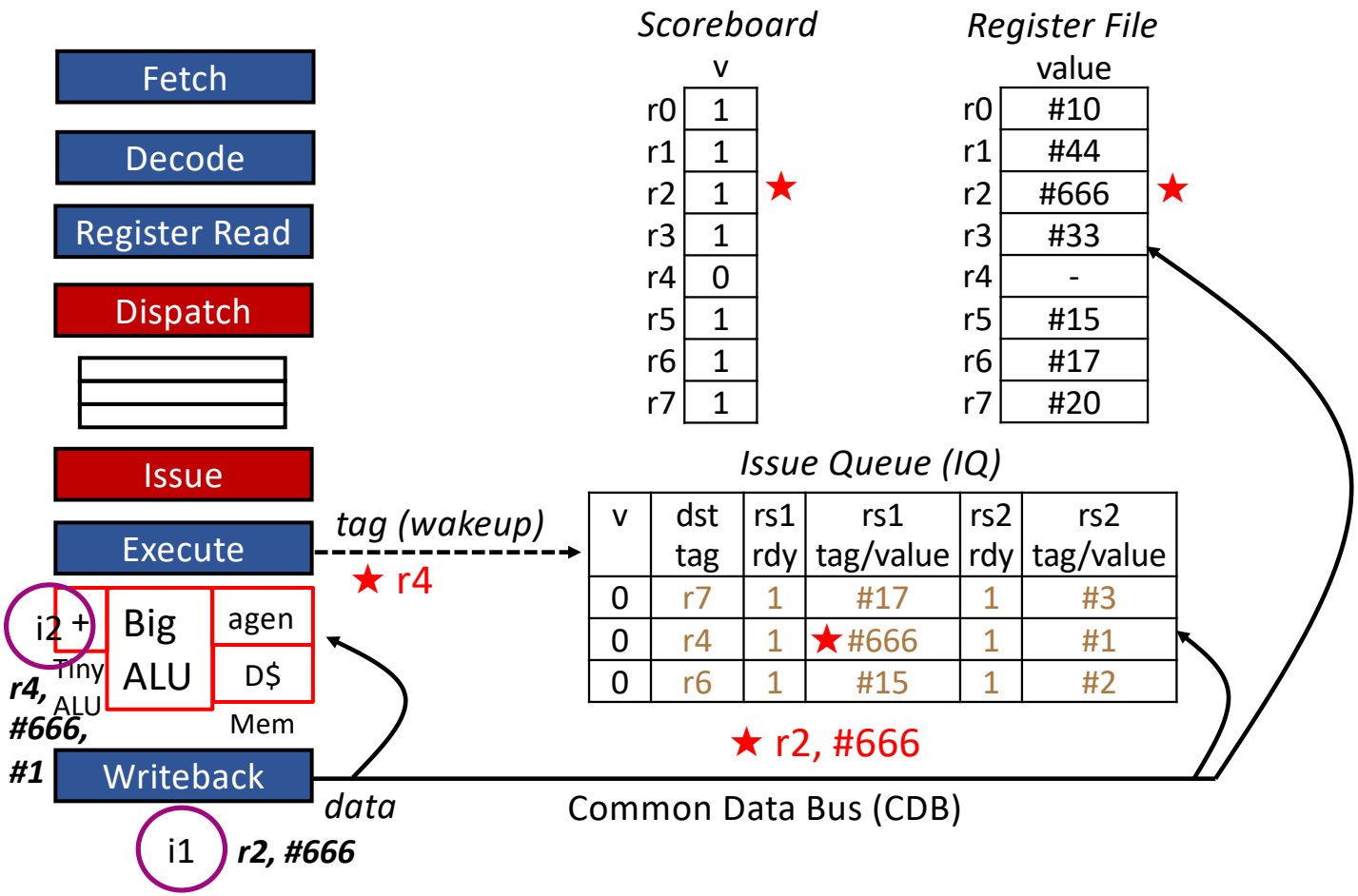
	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
i1: LDR r2, [r1,#0]	FE	DE	RR	DI	IS	$EX_{@}$	$EX_{D\$}$	... miss ...									
i2: ADD r4, r2, #1		FE	DE	RR	DI	IS	IS	IS	IS	IS							
i3: ADD r6, r5, #2			FE	DE	RR	DI	IS	EX	WB								
i4: ADD r7, r6, #3				FE	DE	RR	DI	IS	EX	WB							



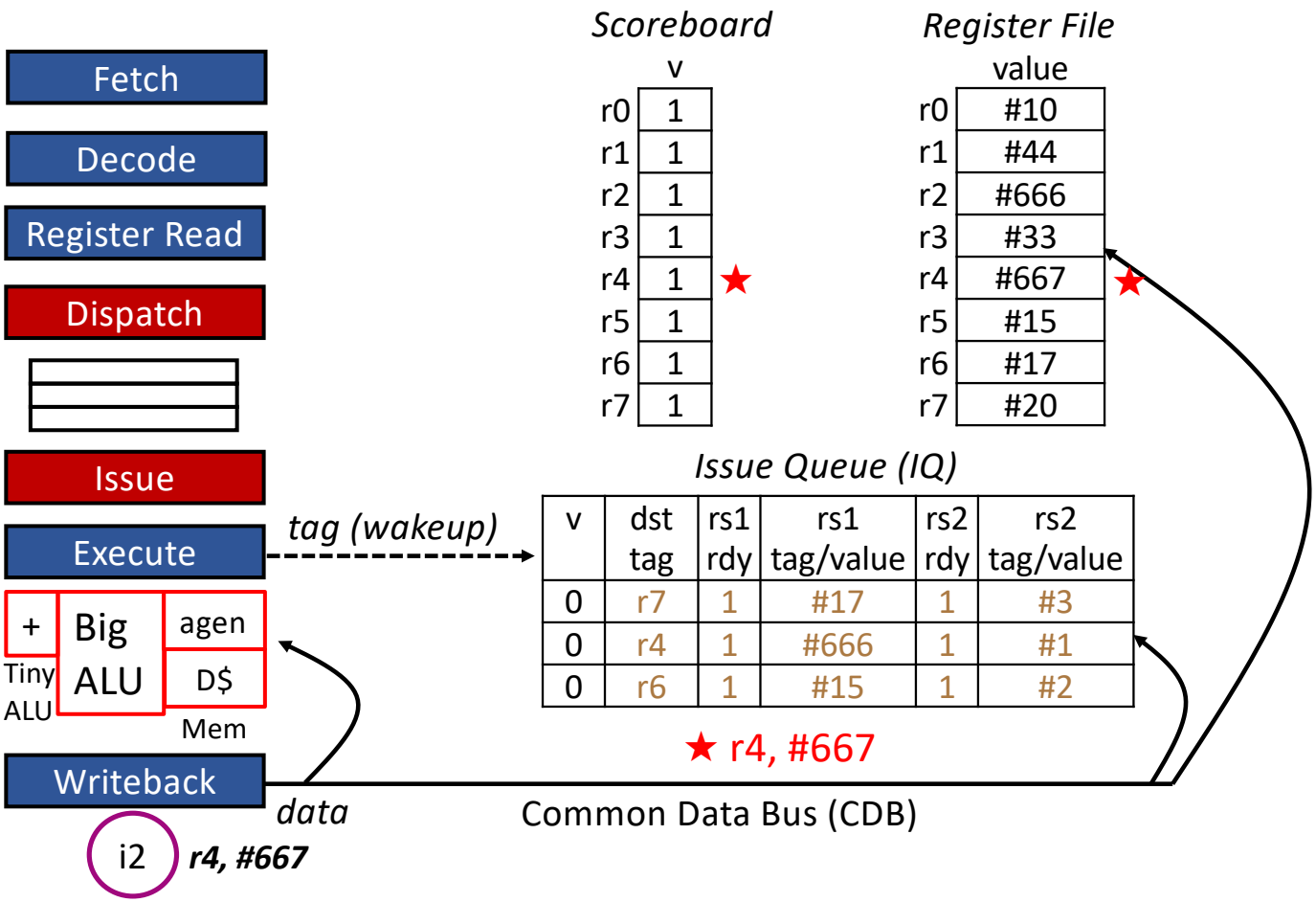
cache miss

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							

What have we studied?

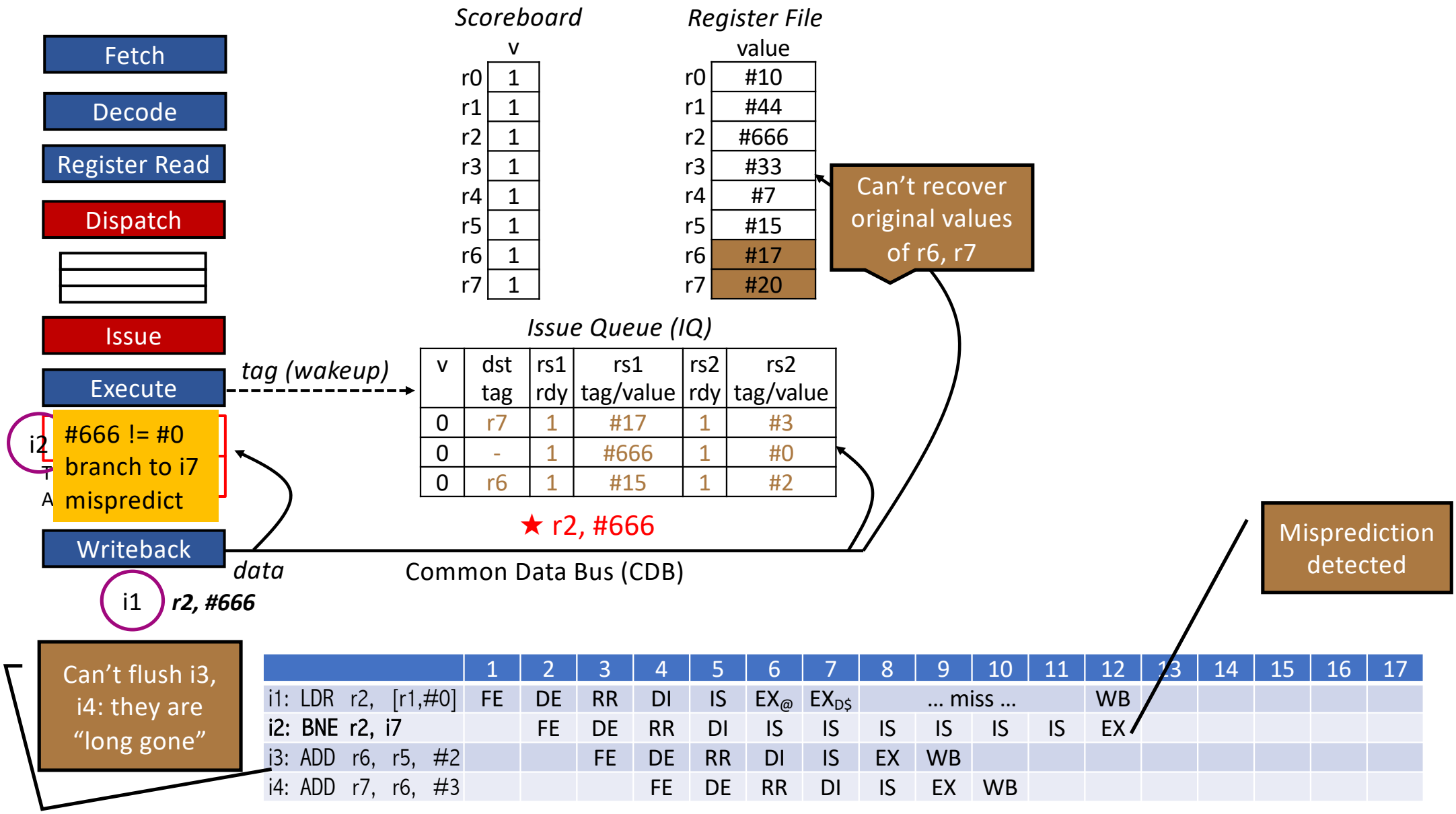
- **Dynamically scheduled superscalar processor**
  - Hardware does “dynamic scheduling” during program execution
  - Can reorder instructions to extract maximum ILP
  - Hardware can construct different “instruction schedules” based on different executions of the same sequence of instructions
    - To 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

Three problems with OOO v.1

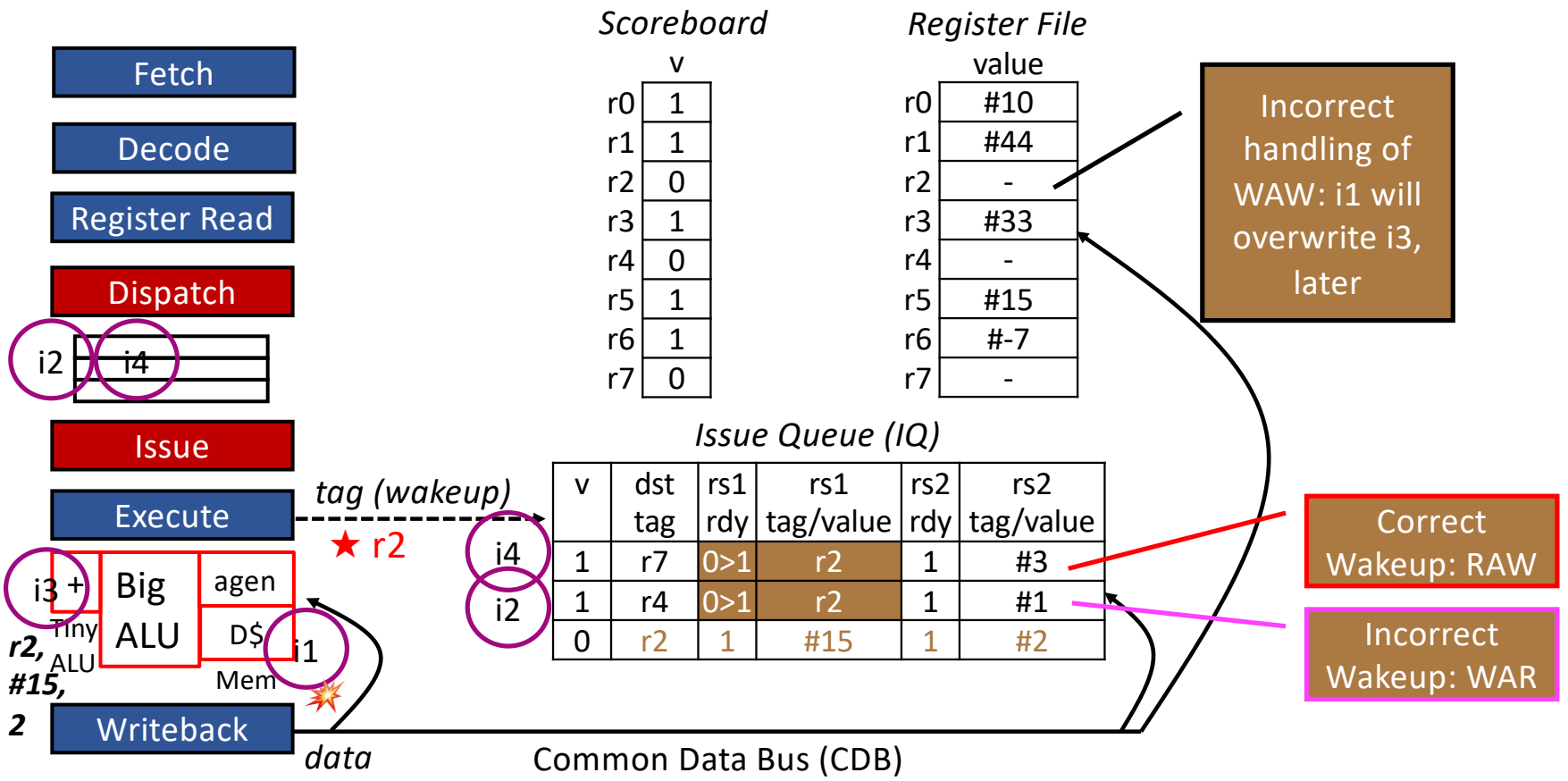
- Cannot recover from misspeculation due to branch misprediction
  - Younger instructions are **speculative** with respect to older instructions
  - Possible to have **older predicted branches** that have not executed yet

- 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



	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: BNE r2, i7		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							



cache miss r2, @44

what happens if we do not stall i3 in RR until i1 executes?

	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									
i3: ADD r2, r5, #2			FE	DE	RR	DI	IS	EX									
i4: ADD r7, r2, #3				FE	DE	RR	DI	IS									

Precise Interrupts/Exceptions

Precise Exception

- **An exception is precise if**
  - All instructions before the faulting (exception causing) instruction have been completed and their effects are visible
  - The faulting instruction has not been completed — it either caused the exception or was not executed at all
  - No later instructions have been executed or made any visible changes to the machine state (e.g., registers or memory)

Precise Exception: Example

- Consider a load instruction that cause a memory protection fault
  - `LDR R1, [R1, #0]`
- In a precise exception system
  - All instructions before this one are completed
  - This instruction causes a fault and is not completed
  - No instructions after this have started execution
  - OS can now handle the protection fault (e.g., fix the address)
  - And resume the program from the `LDR` instruction safely

Imprecise Exception: Example

- In CDC6600, the following scenario is possible
  - LDR is waiting for data from memory (cache miss)
  - i3, i4, and i5 have finished execution out of order

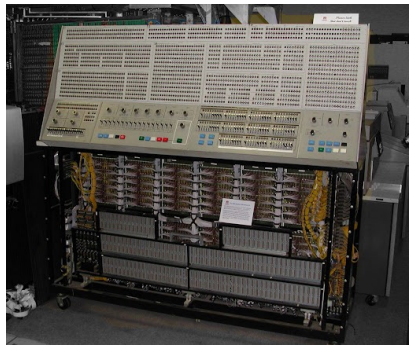
```
i1: ADD   R0,  R0,  #4
i2: LDR   R1,  [R0, #0]
i3: ADD   R2,  R2,  #1
i4: ADD   R3,  R3,  #2
i5: ADD   R4,  R3,  #1
i6: SUB   R5,  R1,  #1
```

  - Later on, LDR generates a software exception (illegal memory address)
  - The CPU branches to the exception handler, setting PC to handler address
  - **When exception handler is run, it should see the architectural state (RF) in a state consistent with the sequential programming model**
    - But the contents of RF reflect that i3, i4, and i5 have finished execution
  - We say that the exception is not precise, and CPU does not implement precise exceptions (or interrupts)

Precise Exceptions

- To implement precise exceptions in a **dynamically scheduled CPU**, we need a mechanism that allows:
  - Out-of-order execution of instructions,
  - But in-order commitment to the architectural state.
  - A new structure called the reorder buffer or ROB enables precise exceptions
  - And recovery from branch misprediction

Hardware Speculation



Based on IBM 360/91

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 branch misprediction (a type of misspeculation)

Two Humps in a Modern Pipeline

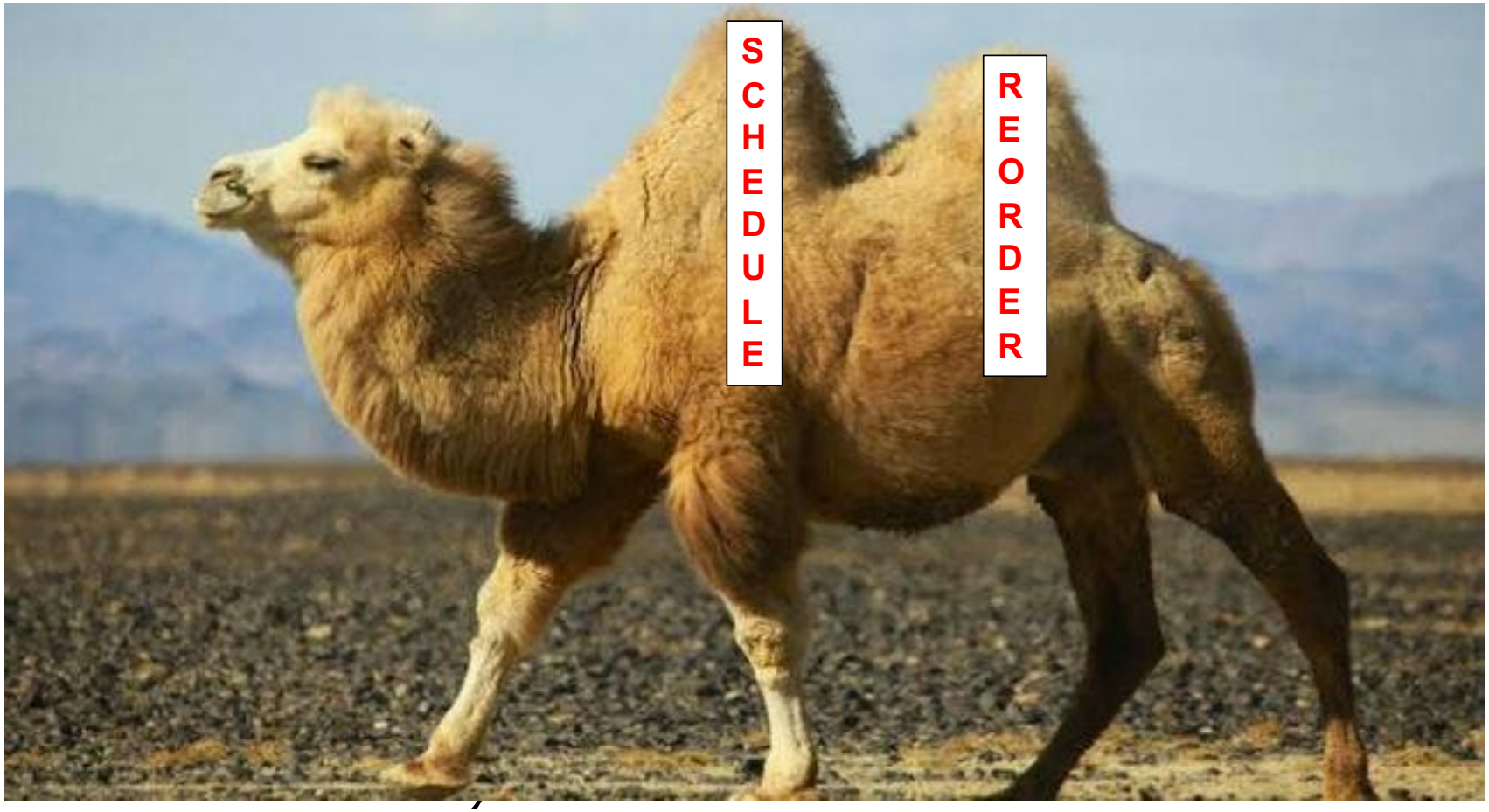
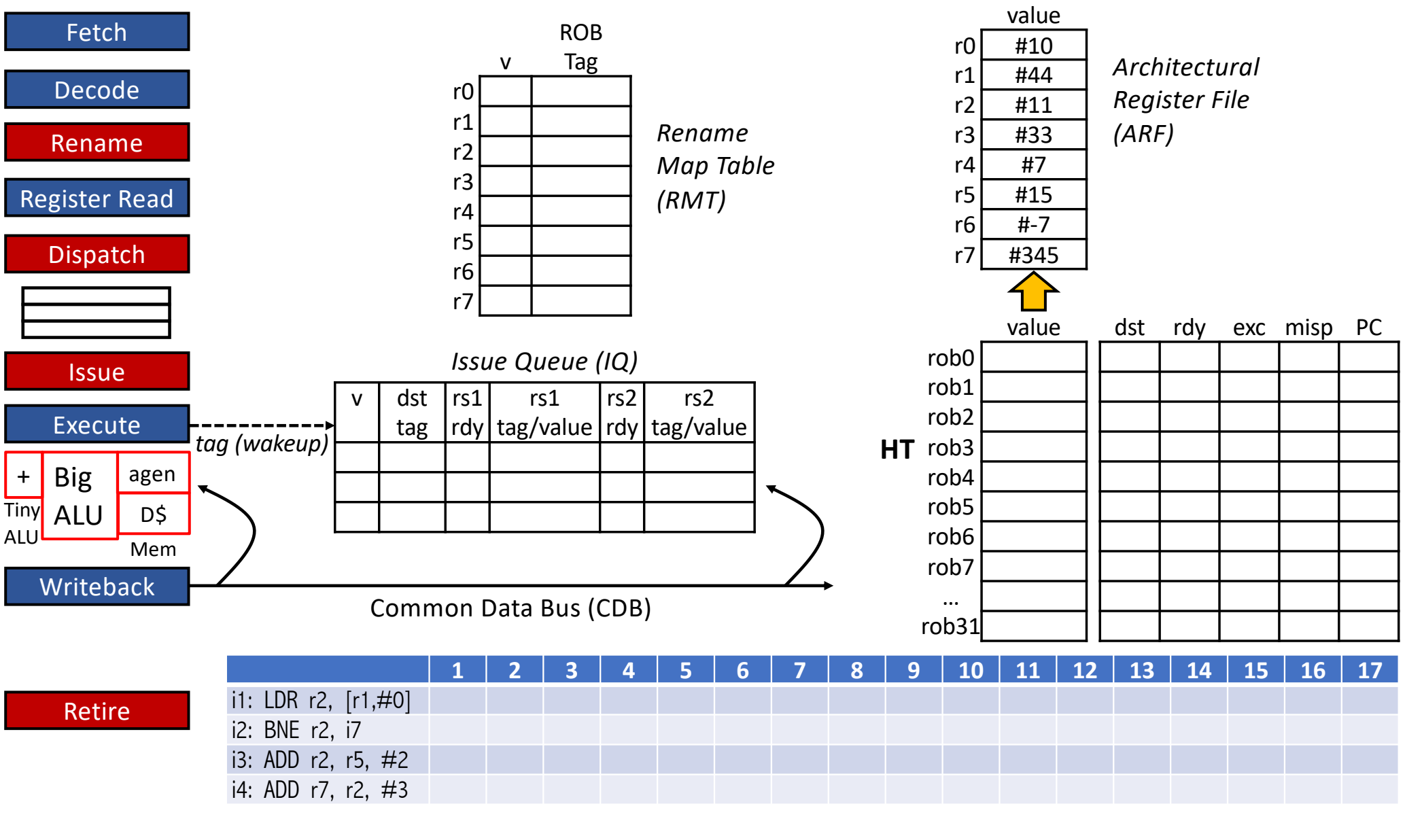


Photo credit: http://true-wildlife.blogspot.ch/2010/10/bactrian-camel.html

Out-of-Order Pipeline (v.2)

- **Solution for WAR/WAW and (im)precise interrupts:** 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



	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
i1: LDR r2, [r1,#0]																	
i2: BNE r2, i7																	
i3: ADD r2, r5, #2																	
i4: ADD r7, r2, #3																	

Renaming Example

- Original sequence to the left. **"Renamed sequence"** is to the right
  - Each destination register is renamed to a unique ROB tag
  - Assuming R7 and R8 in RF are up to date

```
i1: ADD   R0,   R7,   #4
i2: ADD   R1,   R0,   #1
i3: ADD   R0,   R8,   #8
i4: ADD   R2,   R0,   #1
```

```
i1: ADD   ROB0,   R7,    #4
i2: ADD   ROB1,   ROB0,  #1
i3: ADD   ROB2,   R8,    #8
i4: ADD   ROB3,   ROB2,  #1
```

- WAW b/w i1 and i3 is eliminated
- All true dependences are still respected
- ROB0, ROB1, ... are an expanded set of microarchitectural registers
  - They are not visible to the programmer (non-architectural)

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.
    - **ROB is a circular buffer with head (H) and tail (T) pointers**

Register Renaming: Operational Details

- 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 Read

- **Register Read Stage**
  - A consumer instruction obtains its source values from one of three places:
    - **ARF:** if producer of value has retired from ROB
    - **ROB (using renamed source):** if producer of value has executed but not yet retired from ROB
    - **Bypass:** if producer of value has not yet executed
  - If renamed to ROB, ROB may indicate value not ready yet
    - Producer hasn't executed yet
    - Instructions keep renamed source as proxy for value

Writeback, Retirement, and Recovery

- Writeback Stage
  - Instruction writes its speculative result OOO into its ROB entry instead of writing to the register file

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

- Misprediction/exception recovery
  - Offending (mispredicted) instruction writes misprediction or exception bit in its ROB entry out of order
  - CPU waits until offending instruction reaches head of ROB (oldest unretired instruction)
  - **When that happens, CPU flushes all instructions** in pipeline and ROB, and restore RMT to be consistent with an empty pipeline

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

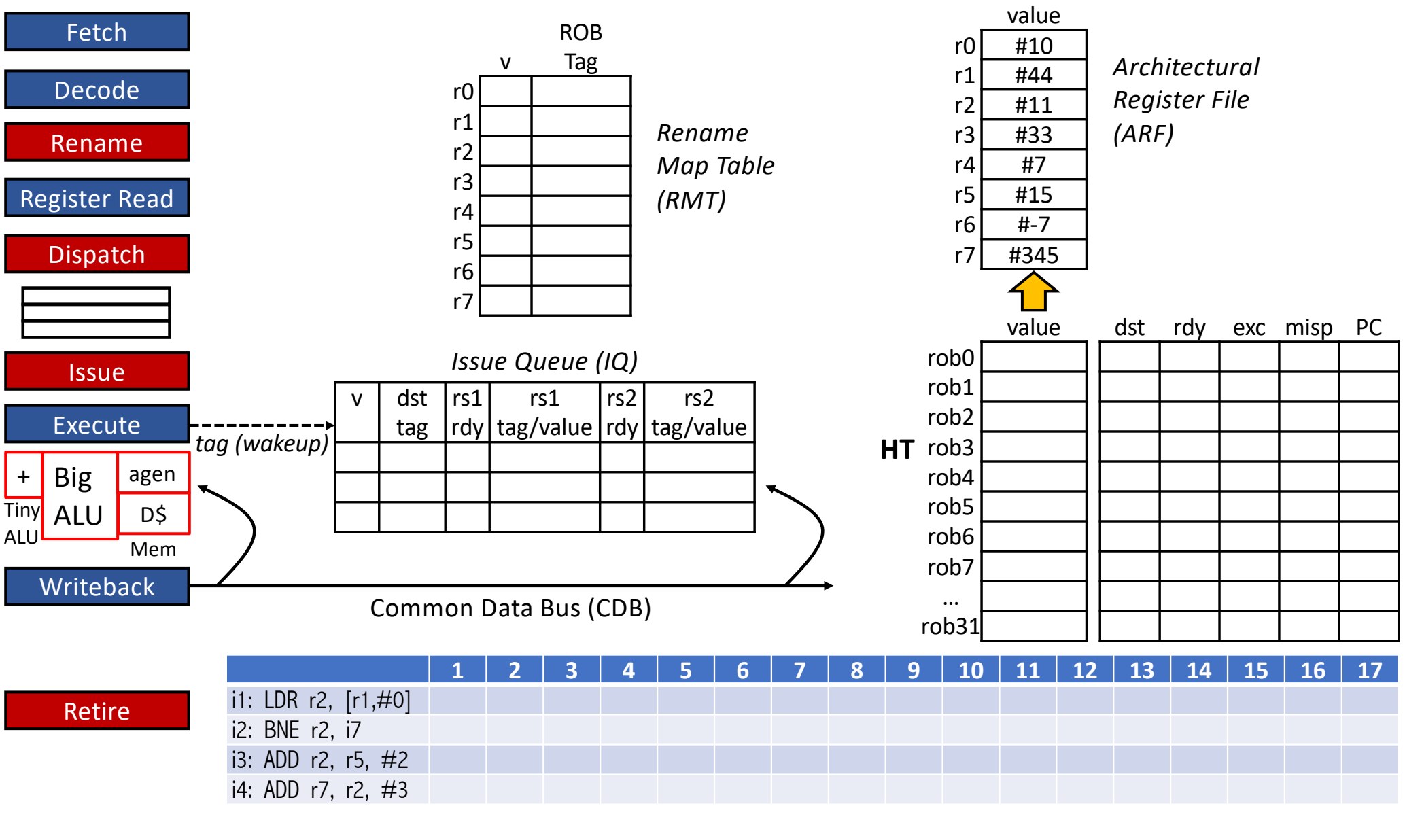
Writeback Stage

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

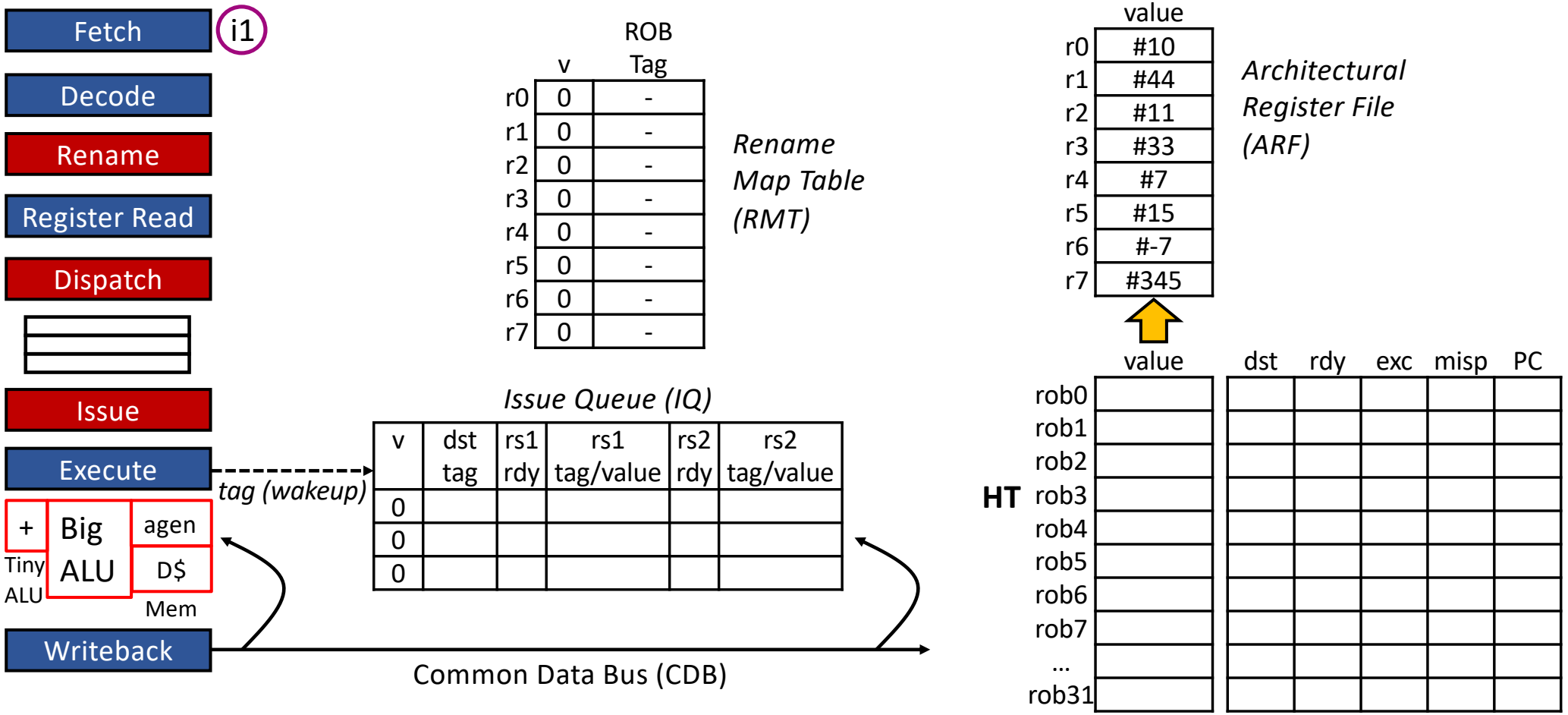
New **Retire** Stage safely commits results from ROB to ARF in program order

Misprediction/exception recovery

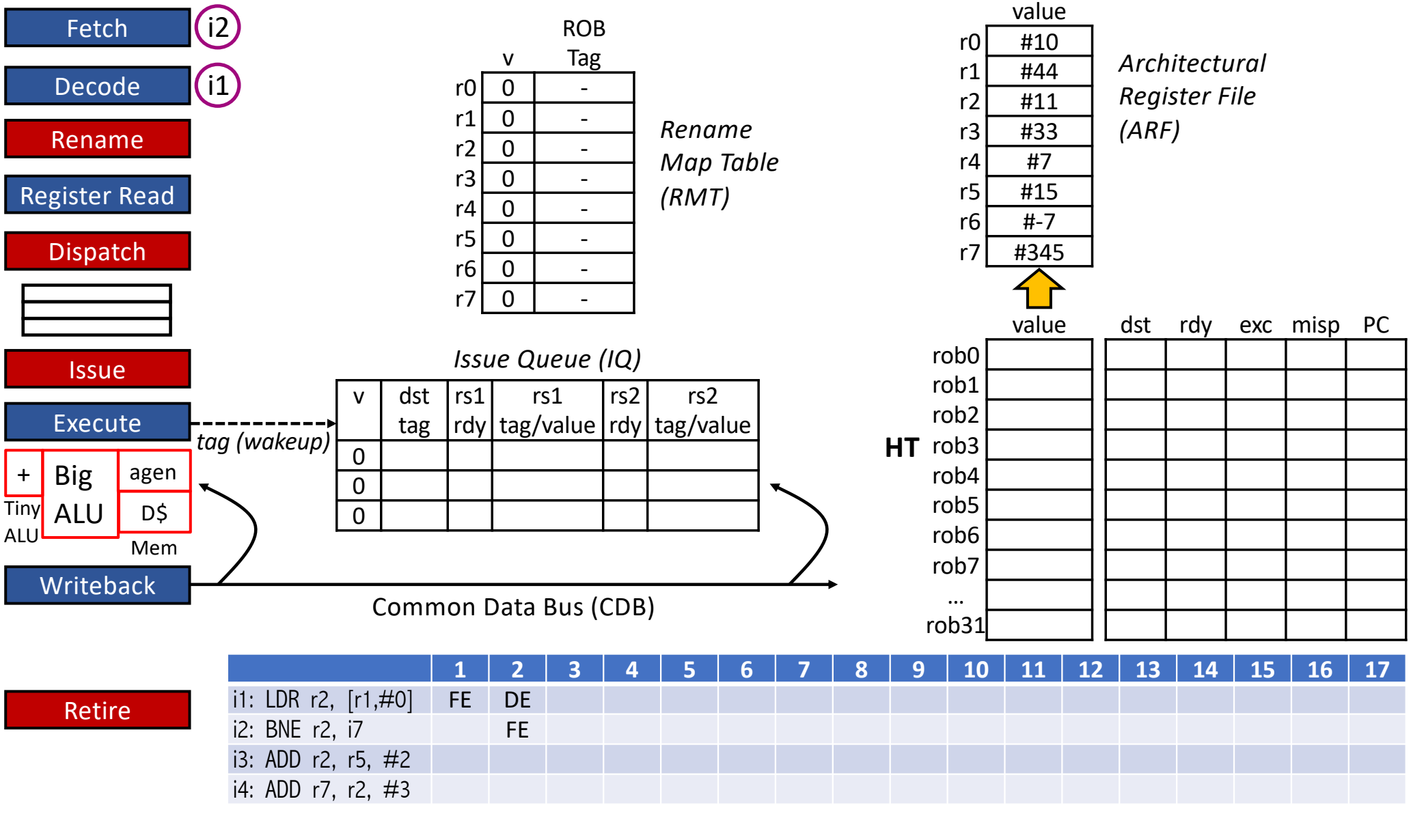
- Offending instruction posts misprediction or exception bit in its ROB entry OOO
- Wait until offending instruction reaches head of ROB (oldest unretired instruction)
- Squash all instructions in pipeline and ROB, and restore RMT to be consistent with an empty pipeline



	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
i1: LDR r2, [r1,#0]																	
i2: BNE r2, i7																	
i3: ADD r2, r5, #2																	
i4: ADD r7, r2, #3																	

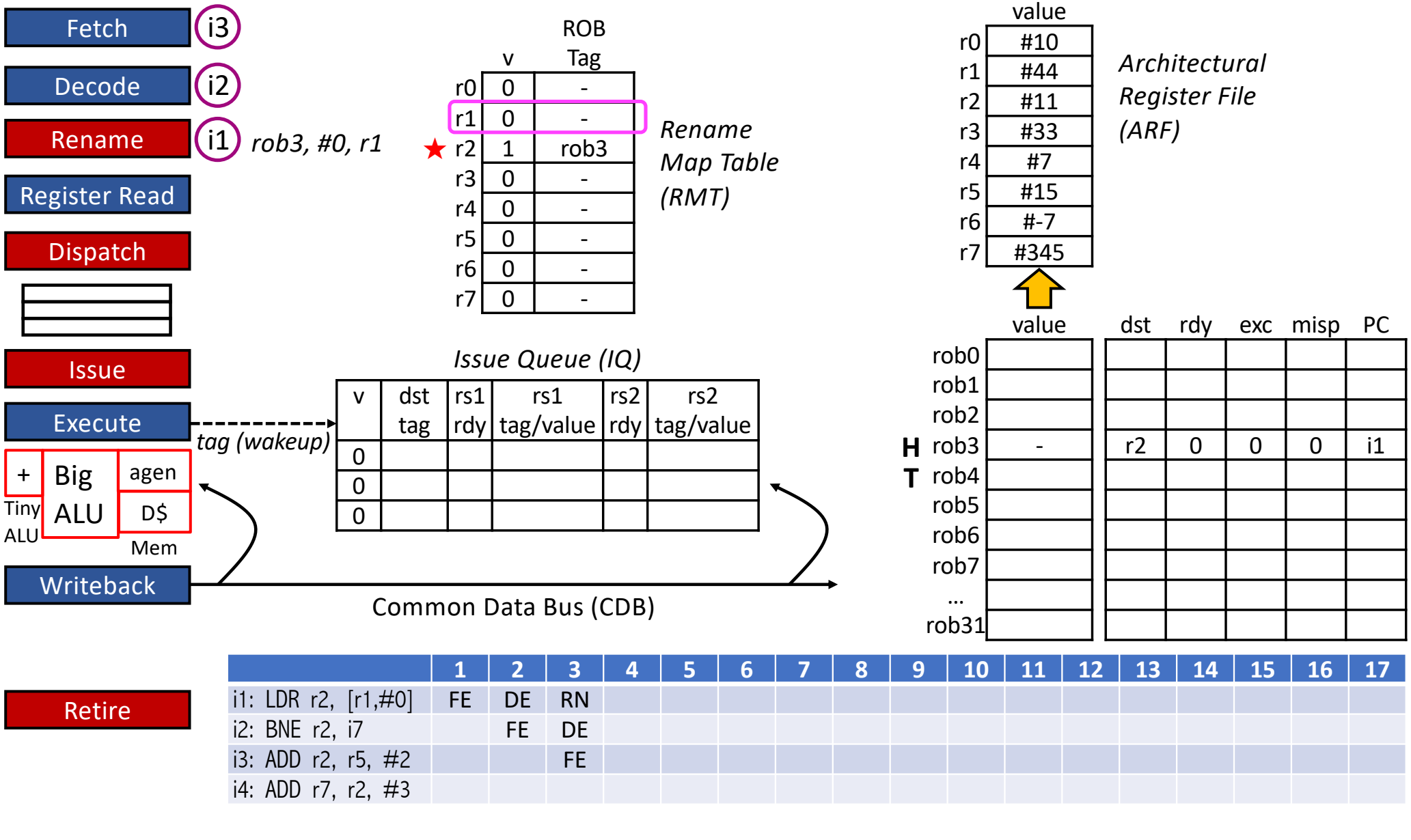


	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
i1: LDR r2, [r1,#0]	FE																
i2: BNE r2, i7																	
i3: ADD r2, r5, #2																	
i4: ADD r7, r2, #3																	

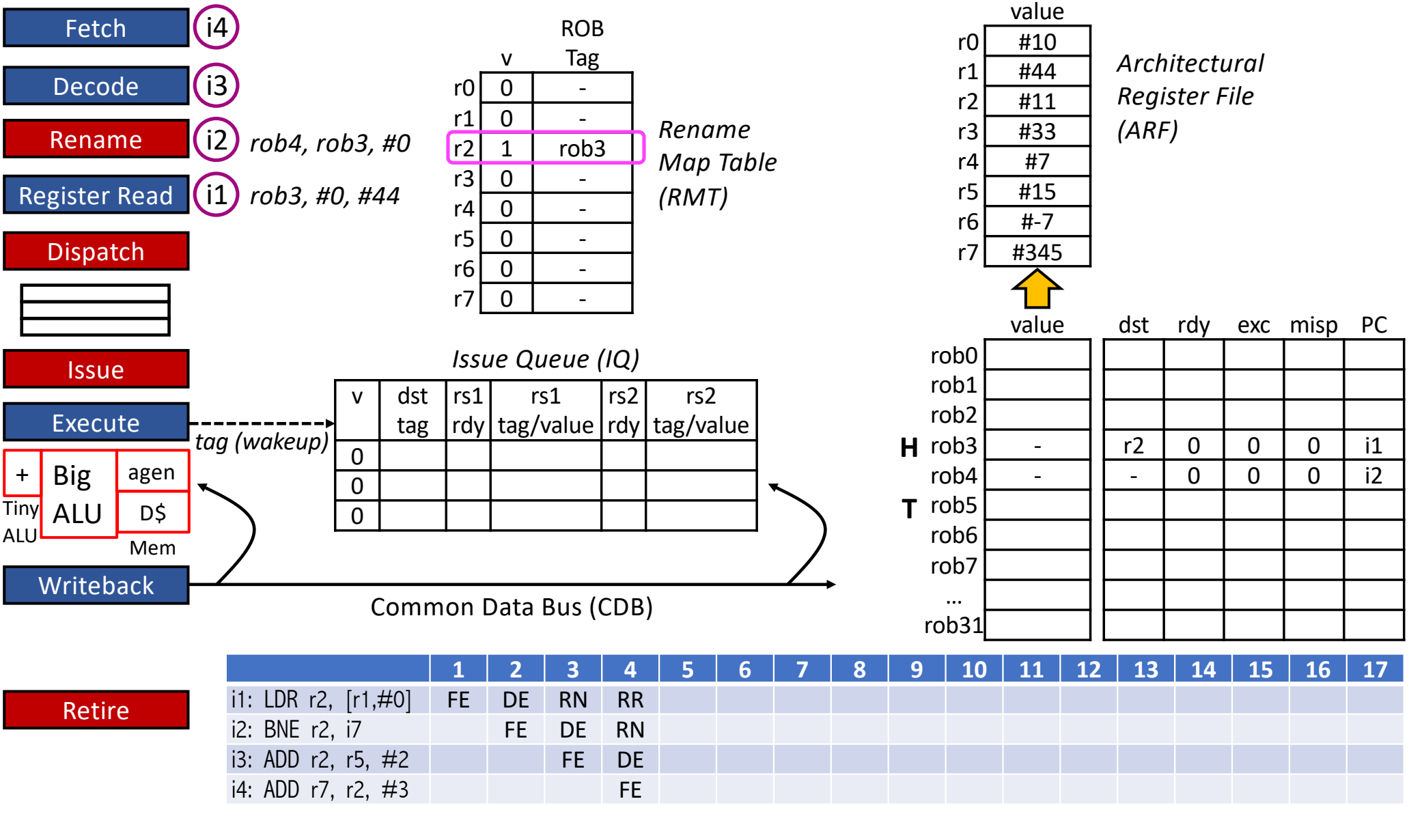


Retire

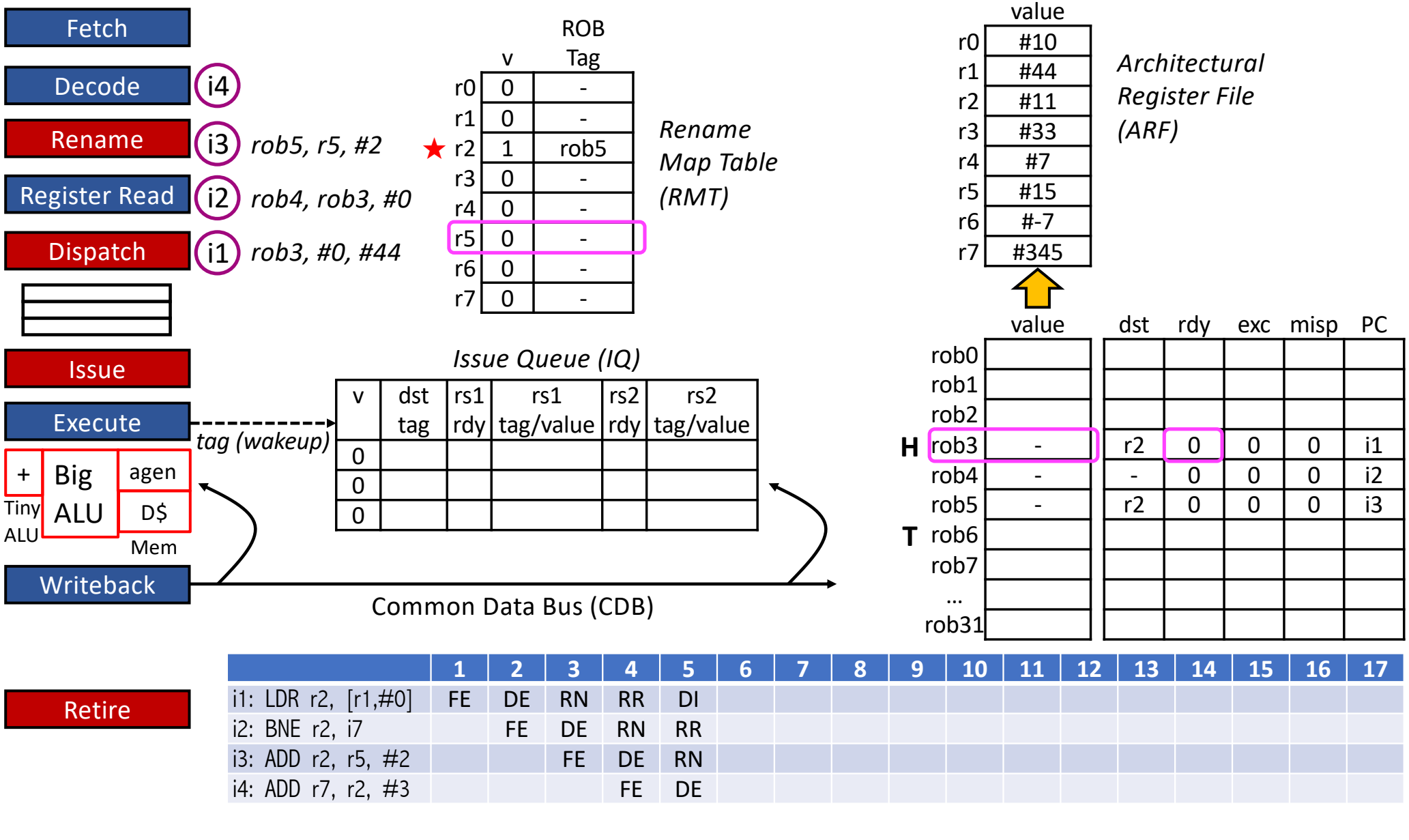
	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
i1: LDR r2, [r1,#0]	FE	DE															
i2: BNE r2, i7		FE															
i3: ADD r2, r5, #2																	
i4: ADD r7, r2, #3																	



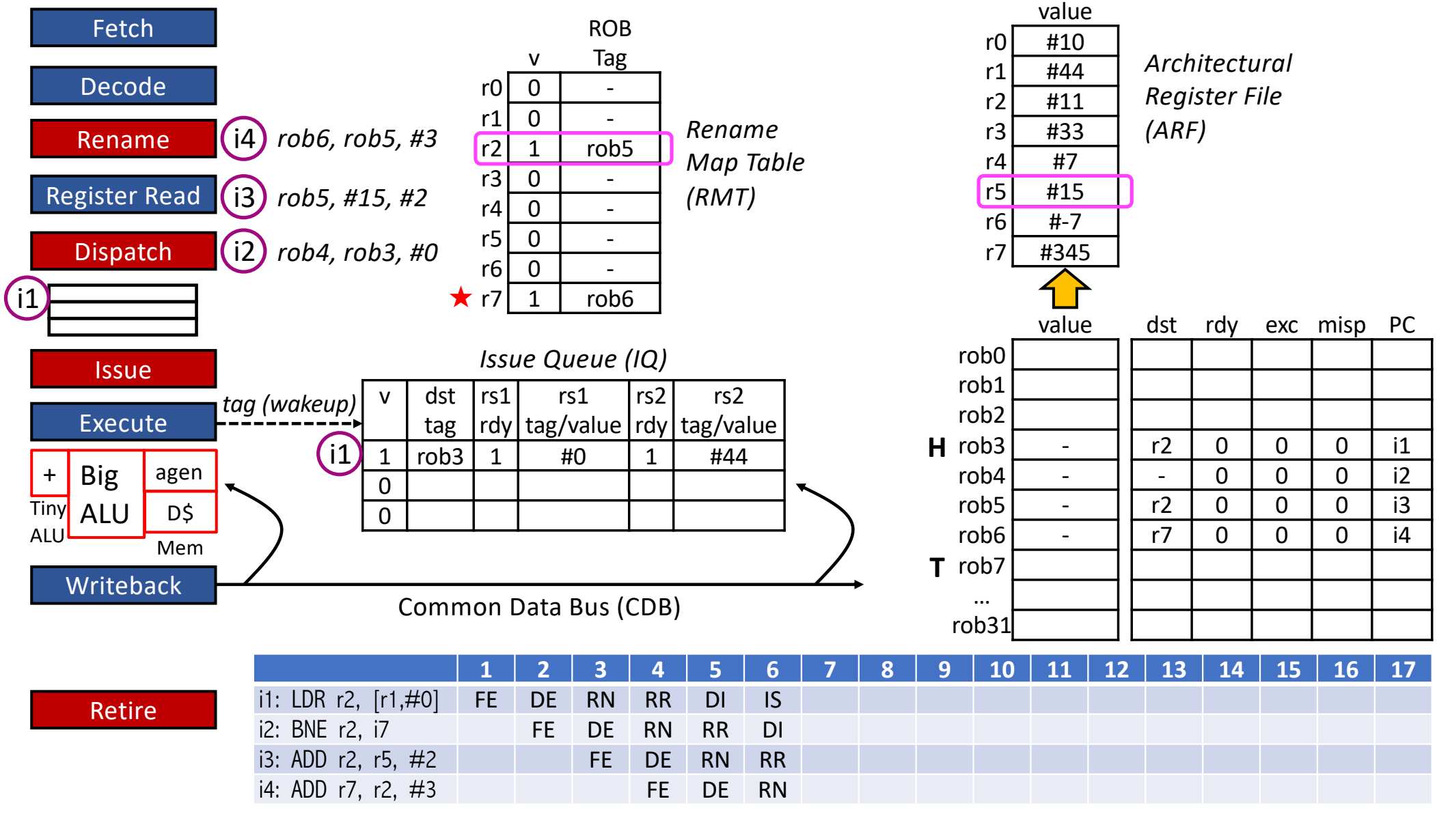
	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
i1: LDR r2, [r1,#0]	FE	DE	RN														
i2: BNE r2, i7		FE	DE														
i3: ADD r2, r5, #2			FE														
i4: ADD r7, r2, #3																	



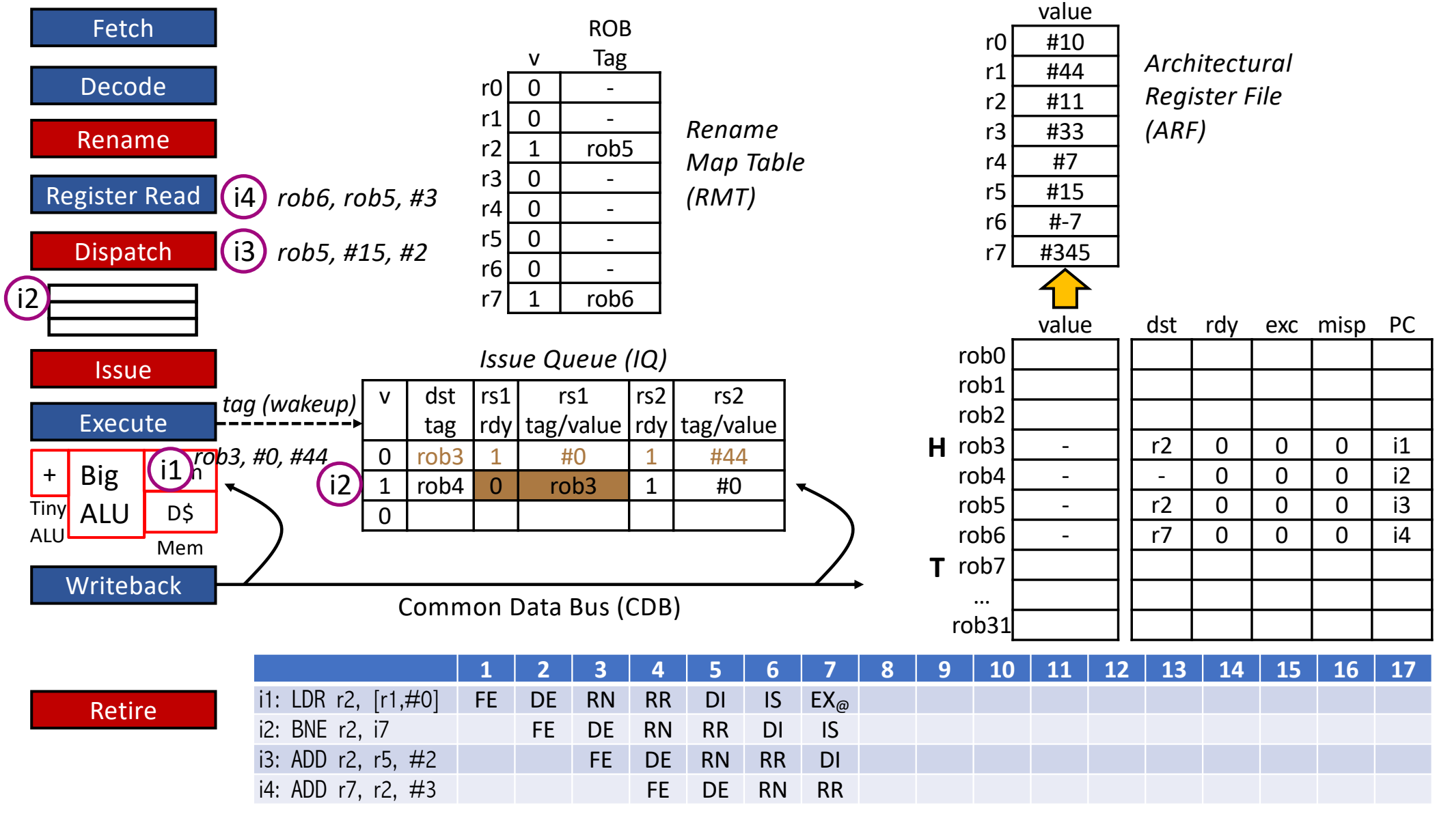
	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
i1: LDR r2, [r1,#0]	FE	DE	RN	RR													
i2: BNE r2, i7		FE	DE	RN													
i3: ADD r2, r5, #2			FE	DE													
i4: ADD r7, r2, #3				FE													



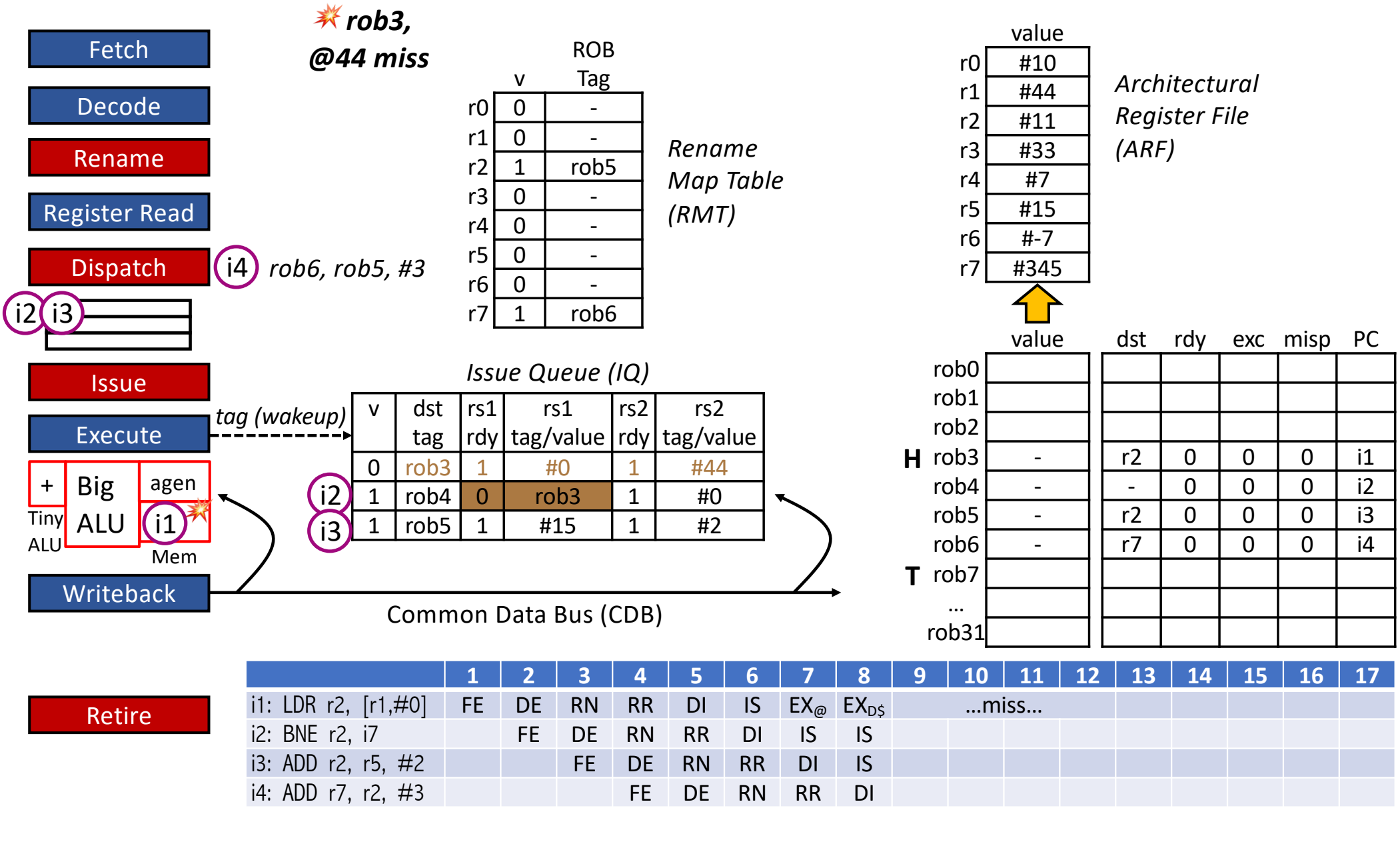
	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
i1: LDR r2, [r1,#0]	FE	DE	RN	RR	DI												
i2: BNE r2, i7		FE	DE	RN	RR												
i3: ADD r2, r5, #2			FE	DE	RN												
i4: ADD r7, r2, #3				FE	DE												



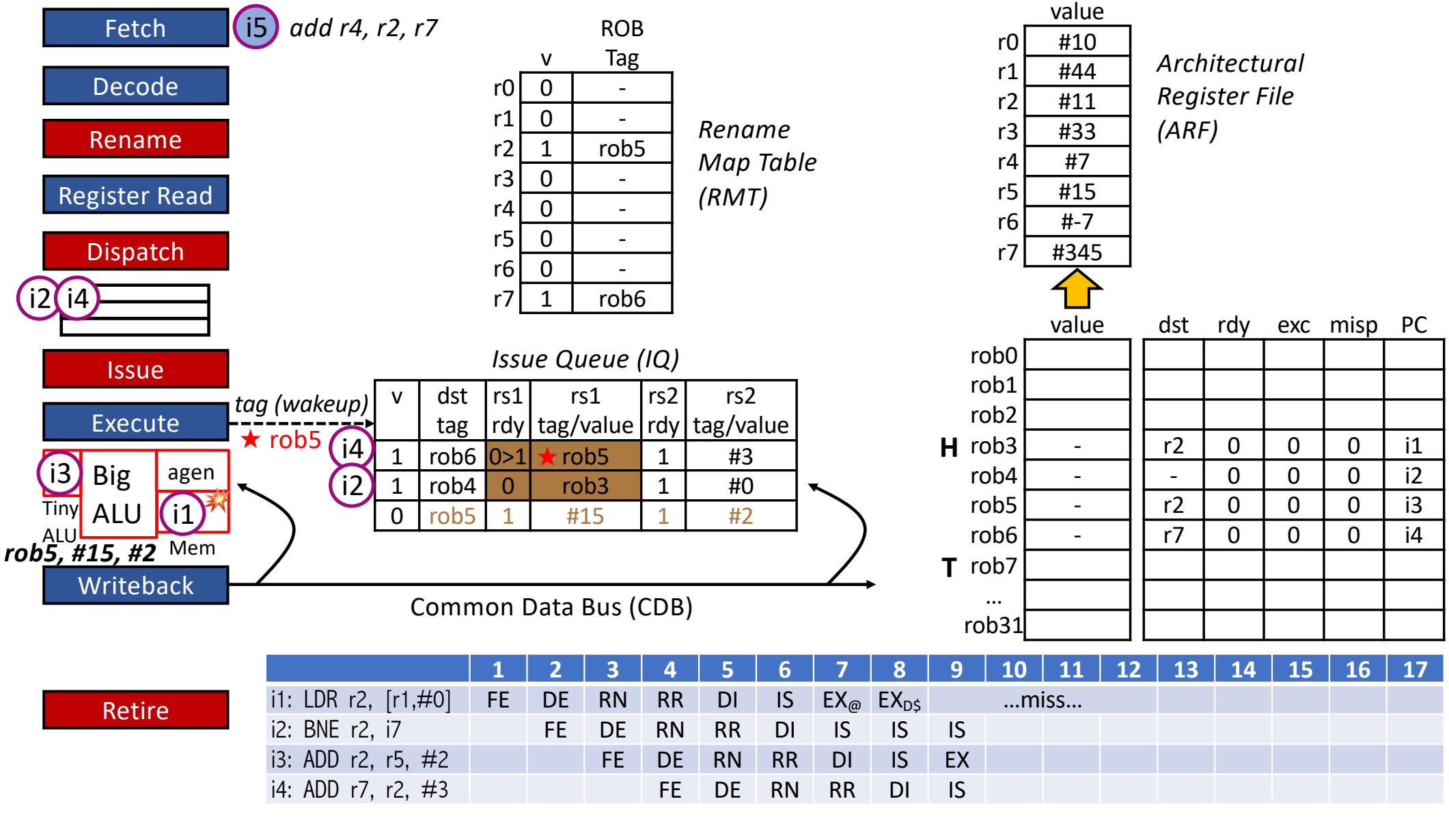
	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
i1: LDR r2, [r1,#0]	FE	DE	RN	RR	DI	IS											
i2: BNE r2, i7		FE	DE	RN	RR	DI											
i3: ADD r2, r5, #2			FE	DE	RN	RR											
i4: ADD r7, r2, #3				FE	DE	RN											



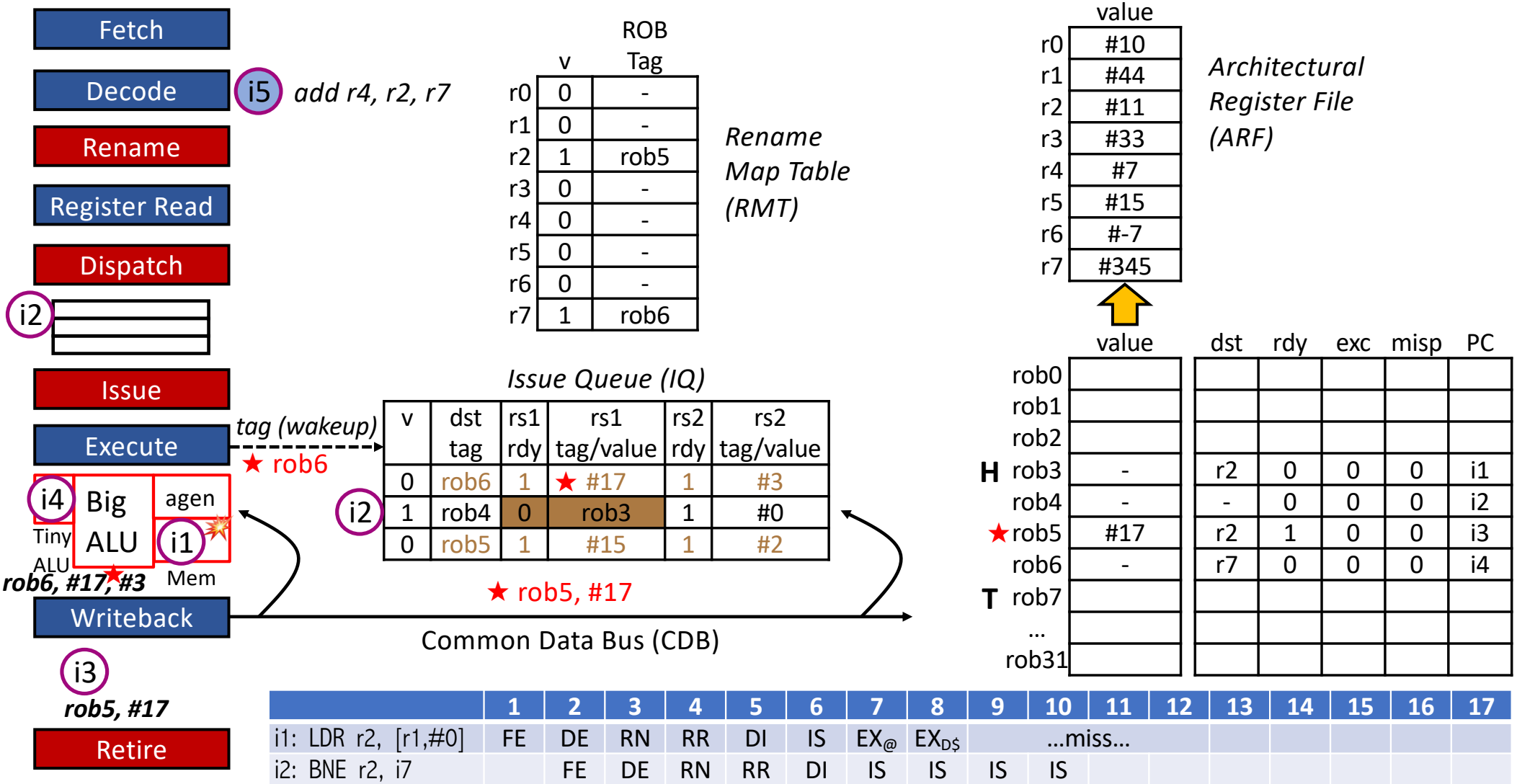
	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
i1: LDR r2, [r1,#0]	FE	DE	RN	RR	DI	IS	$EX_{@}$										
i2: BNE r2, i7		FE	DE	RN	RR	DI	IS										
i3: ADD r2, r5, #2			FE	DE	RN	RR	DI										
i4: ADD r7, r2, #3				FE	DE	RN	RR										



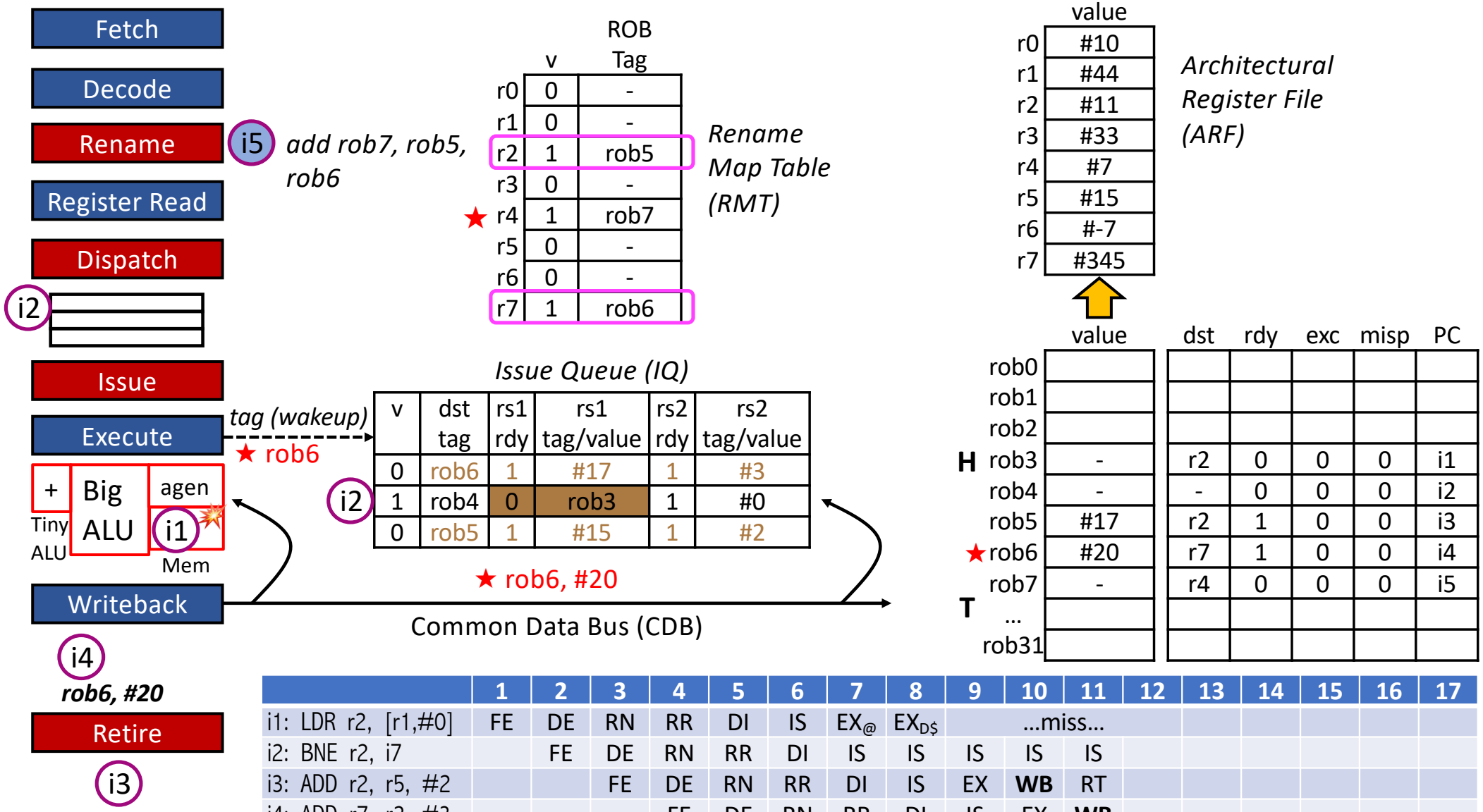
	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
i1: LDR r2, [r1,#0]	FE	DE	RN	RR	DI	IS	$EX_{@}$	$EX_{D\$}$	...miss...								
i2: BNE r2, i7		FE	DE	RN	RR	DI	IS	IS									
i3: ADD r2, r5, #2			FE	DE	RN	RR	DI	IS									
i4: ADD r7, r2, #3				FE	DE	RN	RR	DI									



	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: BNE r2, i7		FE	DE	RN	RR	DI	IS	IS	IS								
i3: ADD r2, r5, #2			FE	DE	RN	RR	DI	IS	EX								
i4: ADD r7, r2, #3				FE	DE	RN	RR	DI	IS								



	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: BNE 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							



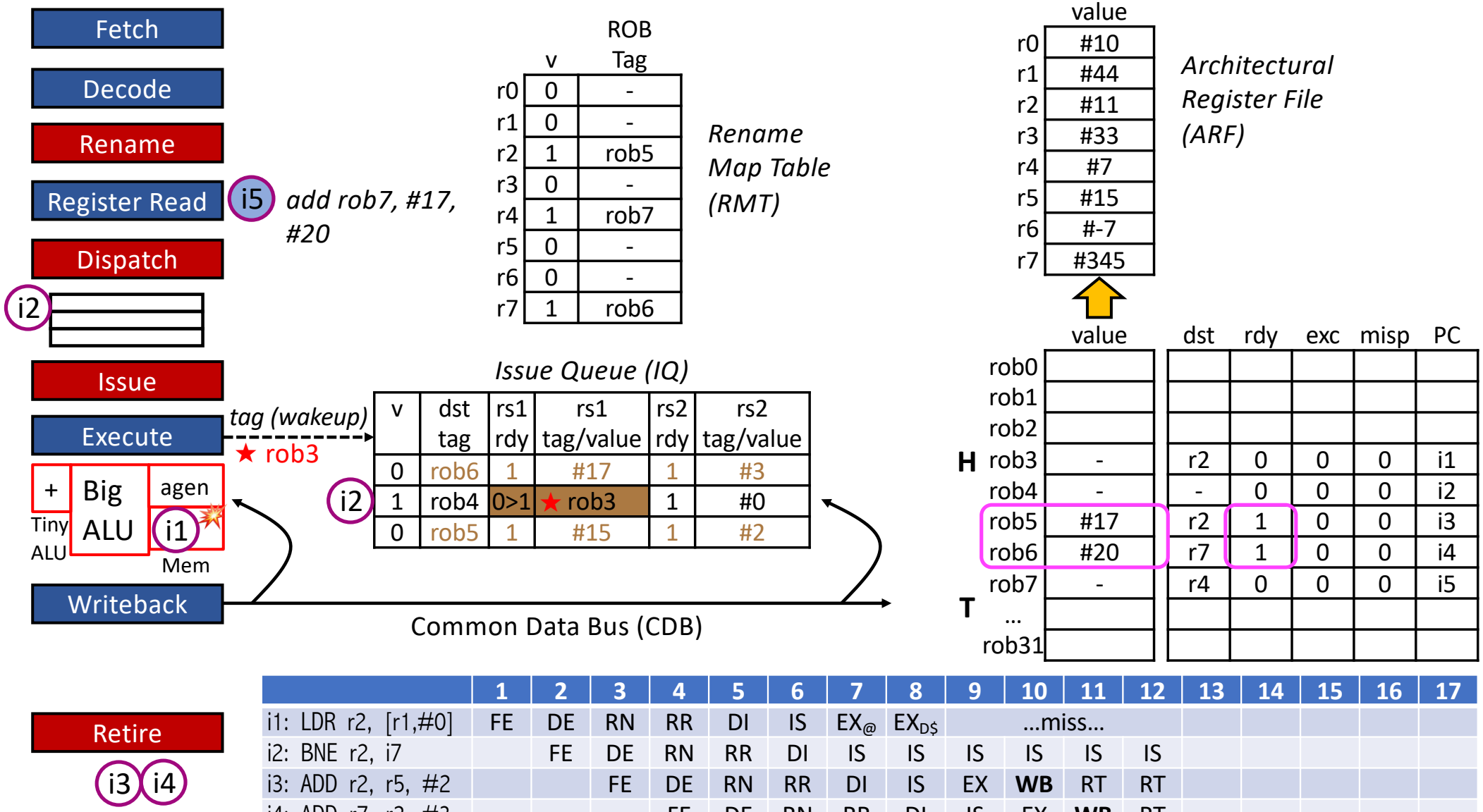
	v	Tag
r0	0	-
r1	0	-
r2	1	rob5
r3	0	-
★ r4	1	rob7
r5	0	-
r6	0	-
r7	1	rob6

	v	dst tag	rs1 rdy	rs1 tag/value	rs2 rdy	rs2 tag/value
	0	rob6	1	#17	1	#3
i2	1	rob4	0	rob3	1	#0
	0	rob5	1	#15	1	#2

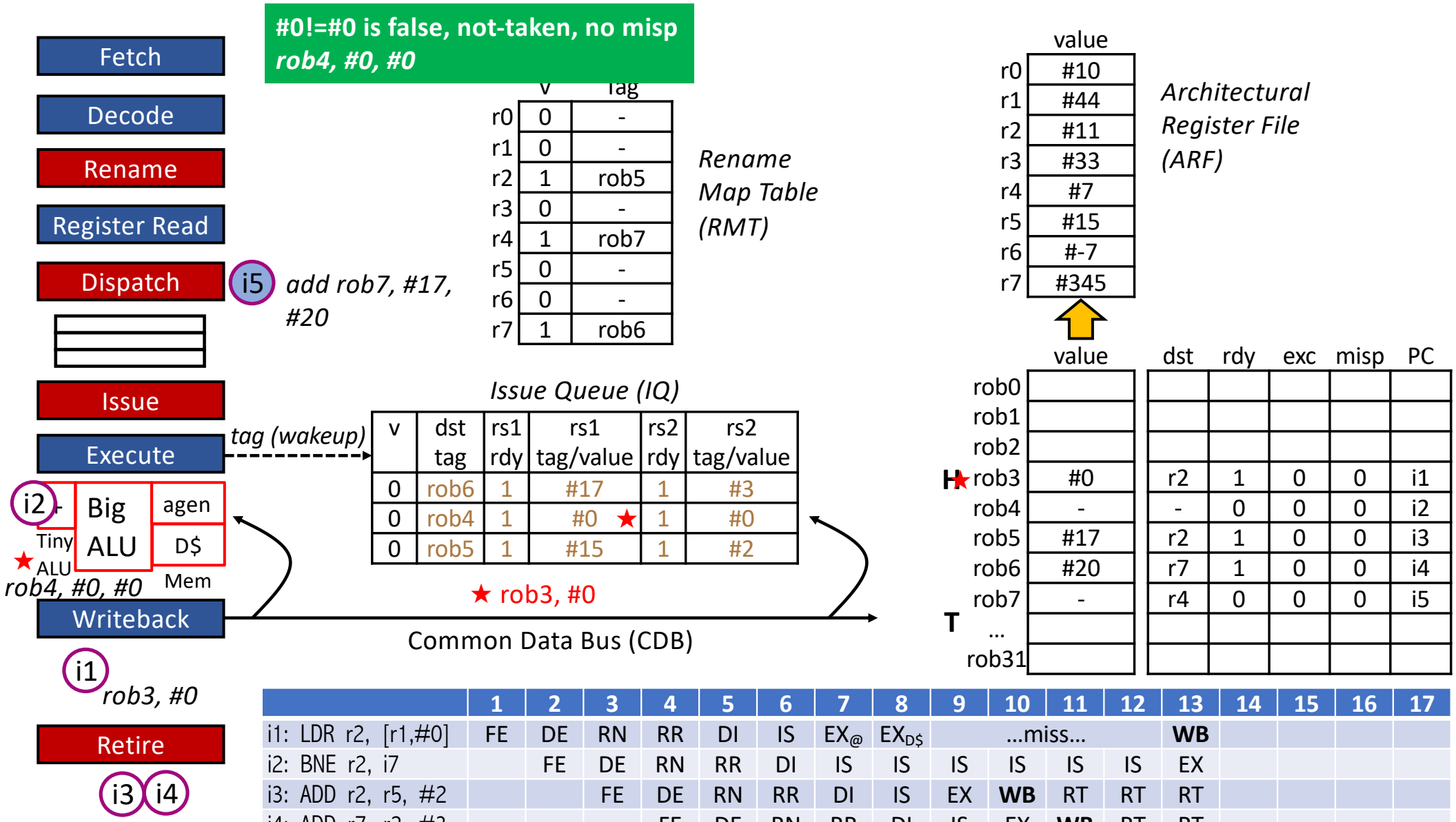
	value
r0	#10
r1	#44
r2	#11
r3	#33
r4	#7
r5	#15
r6	#-7
r7	#345

	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						

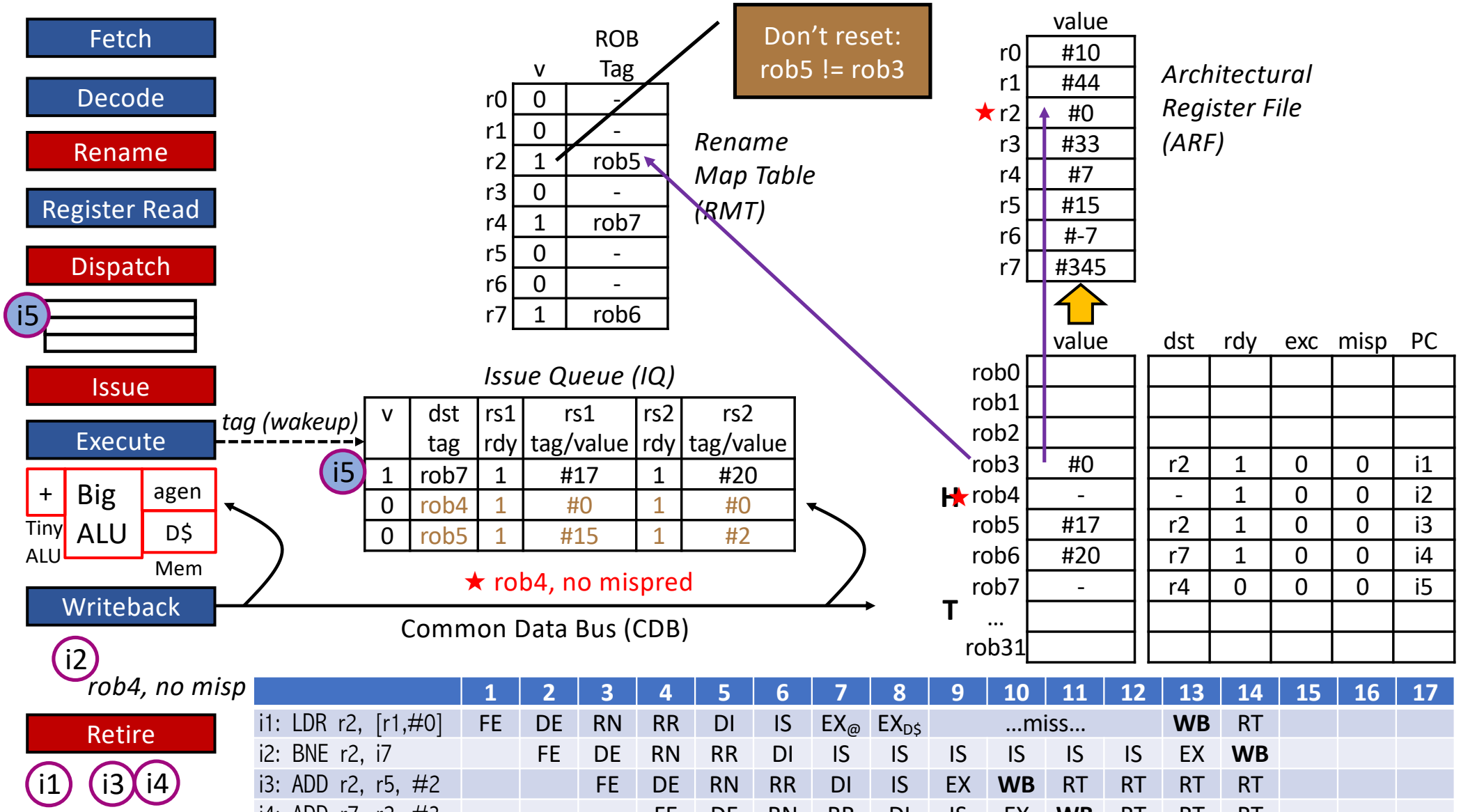
	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
i1: LDR r2, [r1,#0]	FE	DE	RN	RR	DI	IS	$EX_{@}$	$EX_{D\$}$	...miss...								
i2: BNE r2, i7		FE	DE	RN	RR	DI	IS	IS	IS	IS	IS						
i3: ADD r2, r5, #2			FE	DE	RN	RR	DI	IS	EX	**WB**	RT						
i4: ADD r7, r2, #3				FE	DE	RN	RR	DI	IS	EX	**WB**						



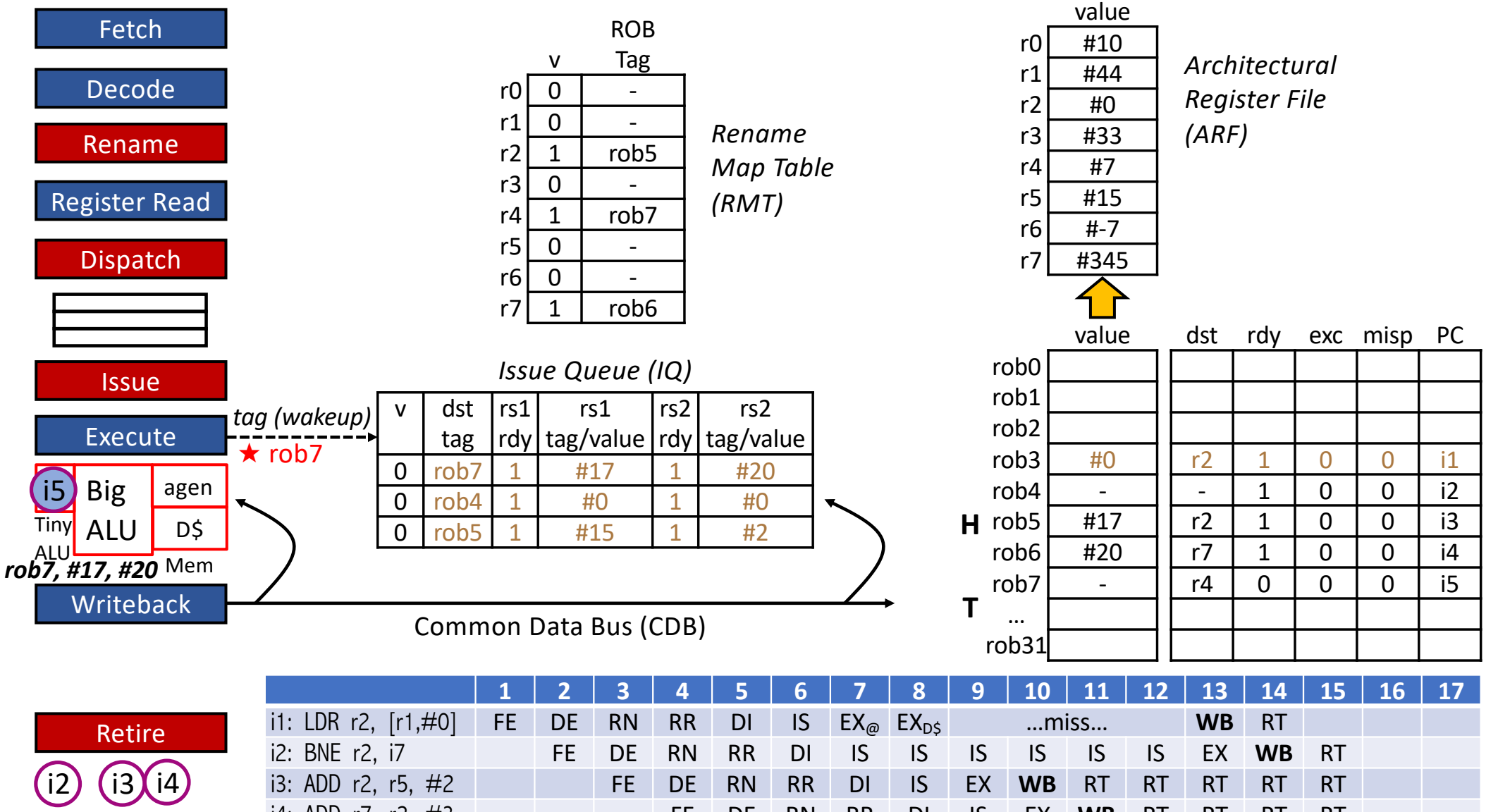
	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
i1: LDR r2, [r1,#0]	FE	DE	RN	RR	DI	IS	$EX_{@}$	$EX_{D\$}$	...miss...								
i2: BNE r2, i7		FE	DE	RN	RR	DI	IS	IS	IS	IS	IS	IS					
i3: ADD r2, r5, #2			FE	DE	RN	RR	DI	IS	EX	**WB**	RT	RT					
i4: ADD r7, r2, #3				FE	DE	RN	RR	DI	IS	EX	**WB**	RT					



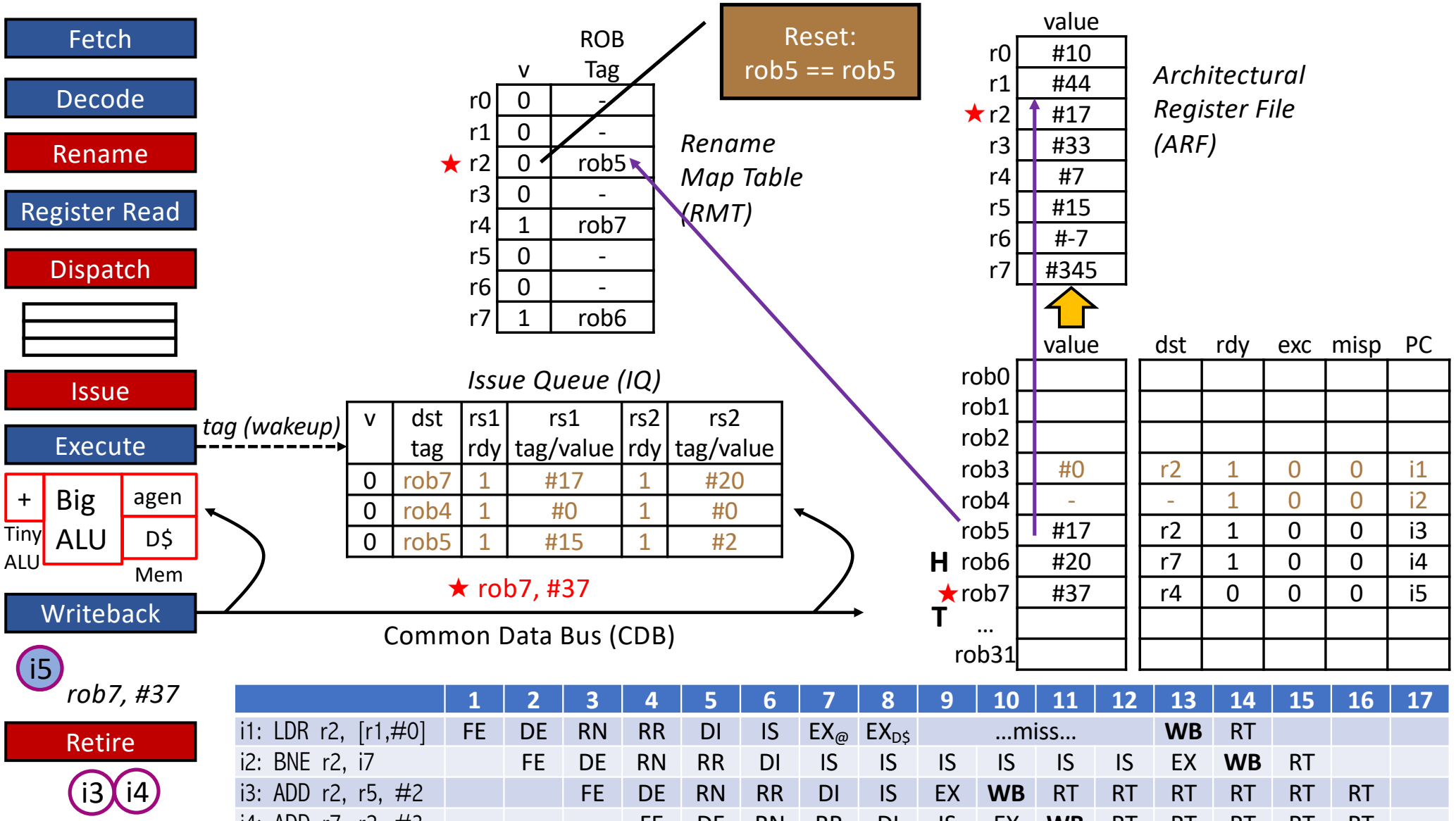
	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
i1: LDR r2, [r1,#0]	FE	DE	RN	RR	DI	IS	$EX_{@}$	$EX_{D\$}$	...miss...				**WB**				
i2: BNE r2, i7		FE	DE	RN	RR	DI	IS	IS	IS	IS	IS	IS	EX				
i3: ADD r2, r5, #2			FE	DE	RN	RR	DI	IS	EX	**WB**	RT	RT	RT				
i4: ADD r7, r2, #3				FE	DE	RN	RR	DI	IS	EX	**WB**	RT	RT				



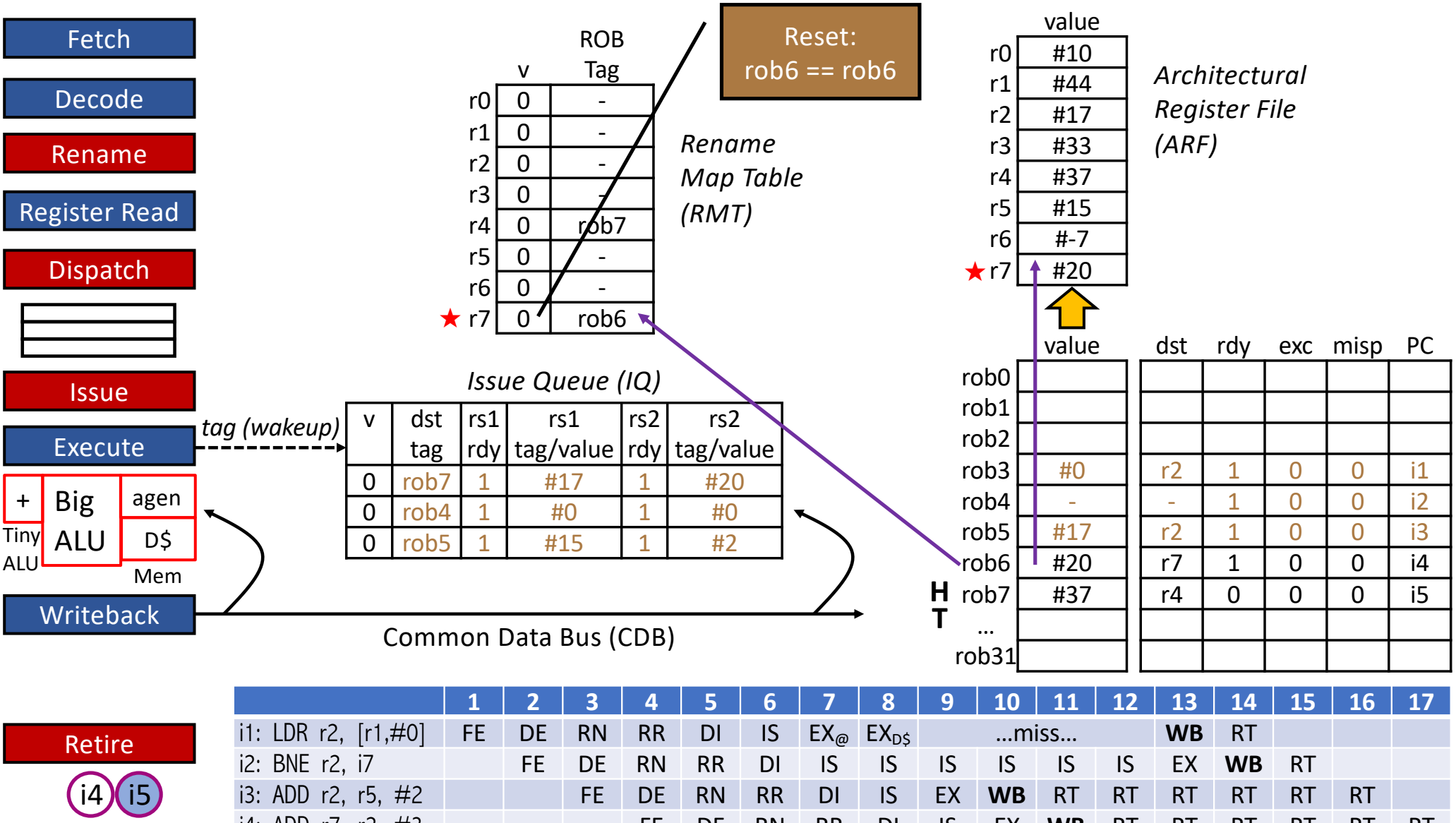
	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
i1: LDR r2, [r1,#0]	FE	DE	RN	RR	DI	IS	$EX_{@}$	$EX_{D\$}$	...miss...				**WB**	RT			
i2: BNE r2, i7		FE	DE	RN	RR	DI	IS	IS	IS	IS	IS	IS	EX	**WB**			
i3: ADD r2, r5, #2			FE	DE	RN	RR	DI	IS	EX	**WB**	RT	RT	RT	RT			
i4: ADD r7, r2, #3				FE	DE	RN	RR	DI	IS	EX	**WB**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



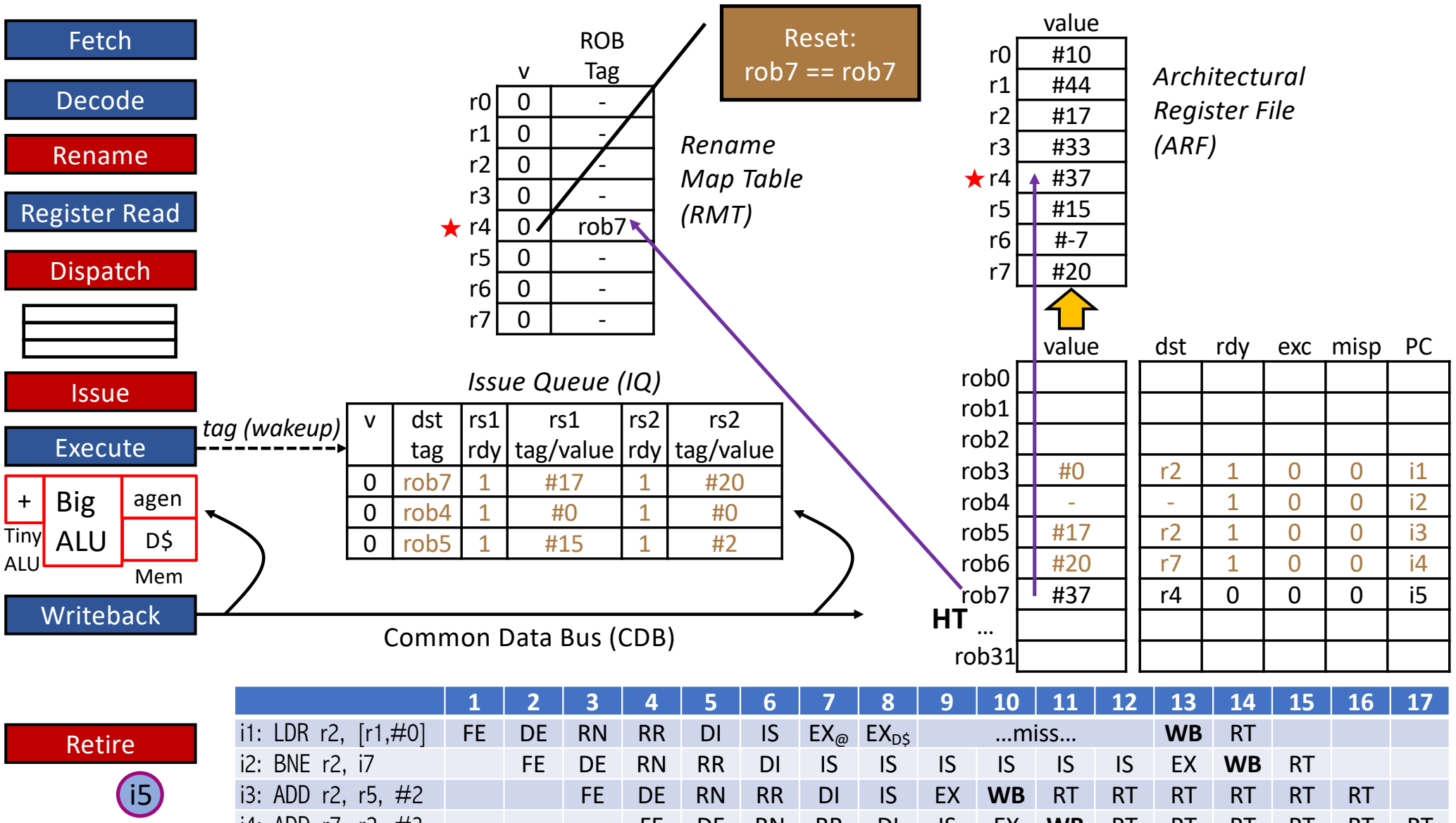
	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
i1: LDR r2, [r1,#0]	FE	DE	RN	RR	DI	IS	$EX_{@}$	$EX_{D\$}$	...miss...				**WB**	RT			
i2: BNE r2, i7		FE	DE	RN	RR	DI	IS	IS	IS	IS	IS	IS	EX	**WB**	RT		
i3: ADD r2, r5, #2			FE	DE	RN	RR	DI	IS	EX	**WB**	RT	RT	RT	RT	RT		
i4: ADD r7, r2, #3				FE	DE	RN	RR	DI	IS	EX	**WB**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



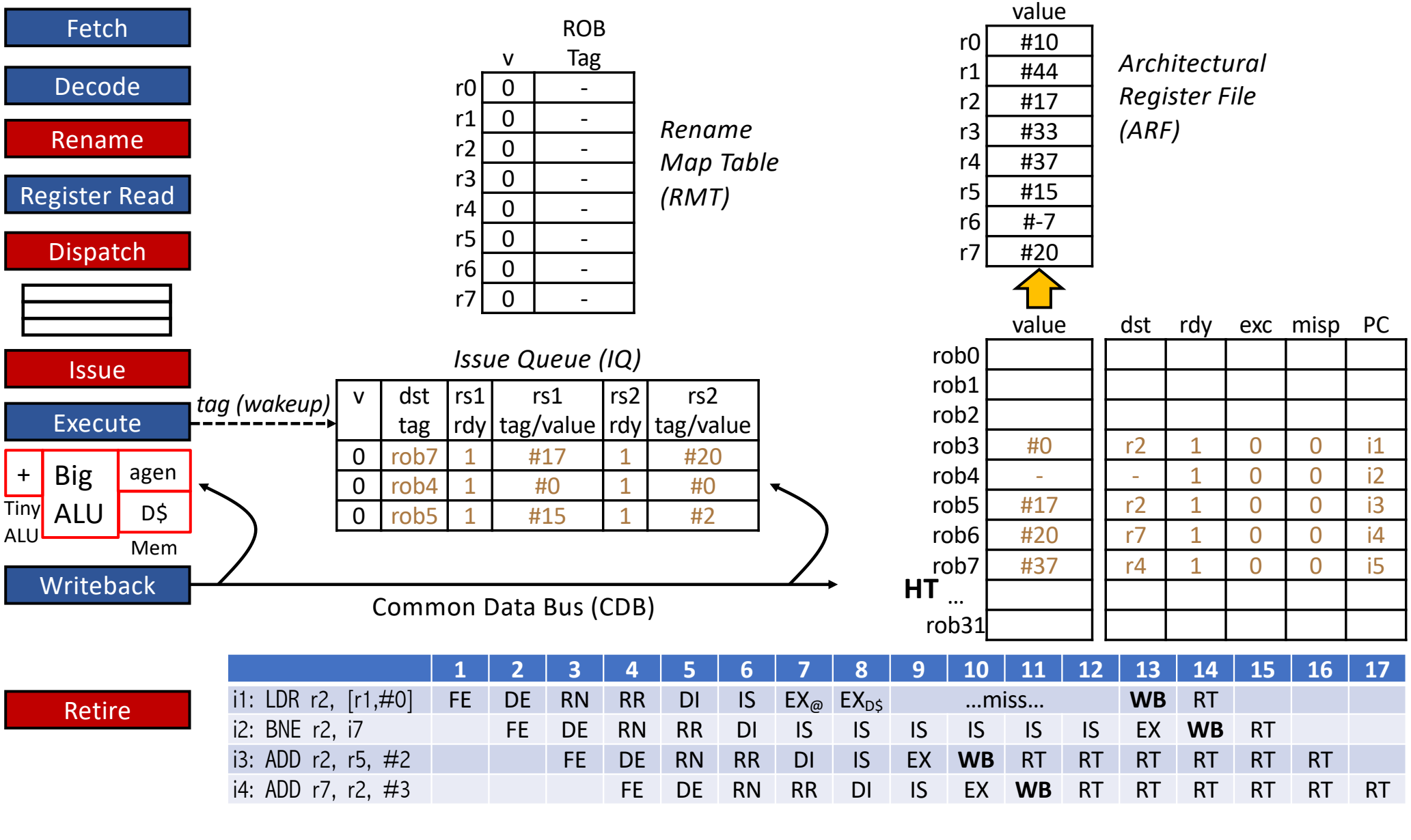
	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
i1: LDR r2, [r1,#0]	FE	DE	RN	RR	DI	IS	$EX_{@}$	$EX_{D\$}$	...miss...				**WB**	RT			
i2: BNE r2, i7		FE	DE	RN	RR	DI	IS	IS	IS	IS	IS	IS	EX	**WB**	RT		
i3: ADD r2, r5, #2			FE	DE	RN	RR	DI	IS	EX	**WB**	RT	RT	RT	RT	RT	RT	
i4: ADD r7, r2, #3				FE	DE	RN	RR	DI	IS	EX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



	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: BNE 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



	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: BNE 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



	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: BNE 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

- `i1 (Rename)`
  1. Allocate entry for `i1` in ROB at `rob3`
     - Tail of ROB is at `rob3`
  2. Rename the destination operand (`r2`) to `rob3`
  3. Increment the tail pointer of ROB to `rob4`
  4. Set `v[r2]=1` in RMT
  5. One source operand is a constant 0
  6. Rename the second source operand `r1` to `ARF[r1]` because in RMT: `v[r1]=0`
- `i2 (Decode)`
- `i3 (Fetch)`
  - The fetch is speculative as `i2` 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

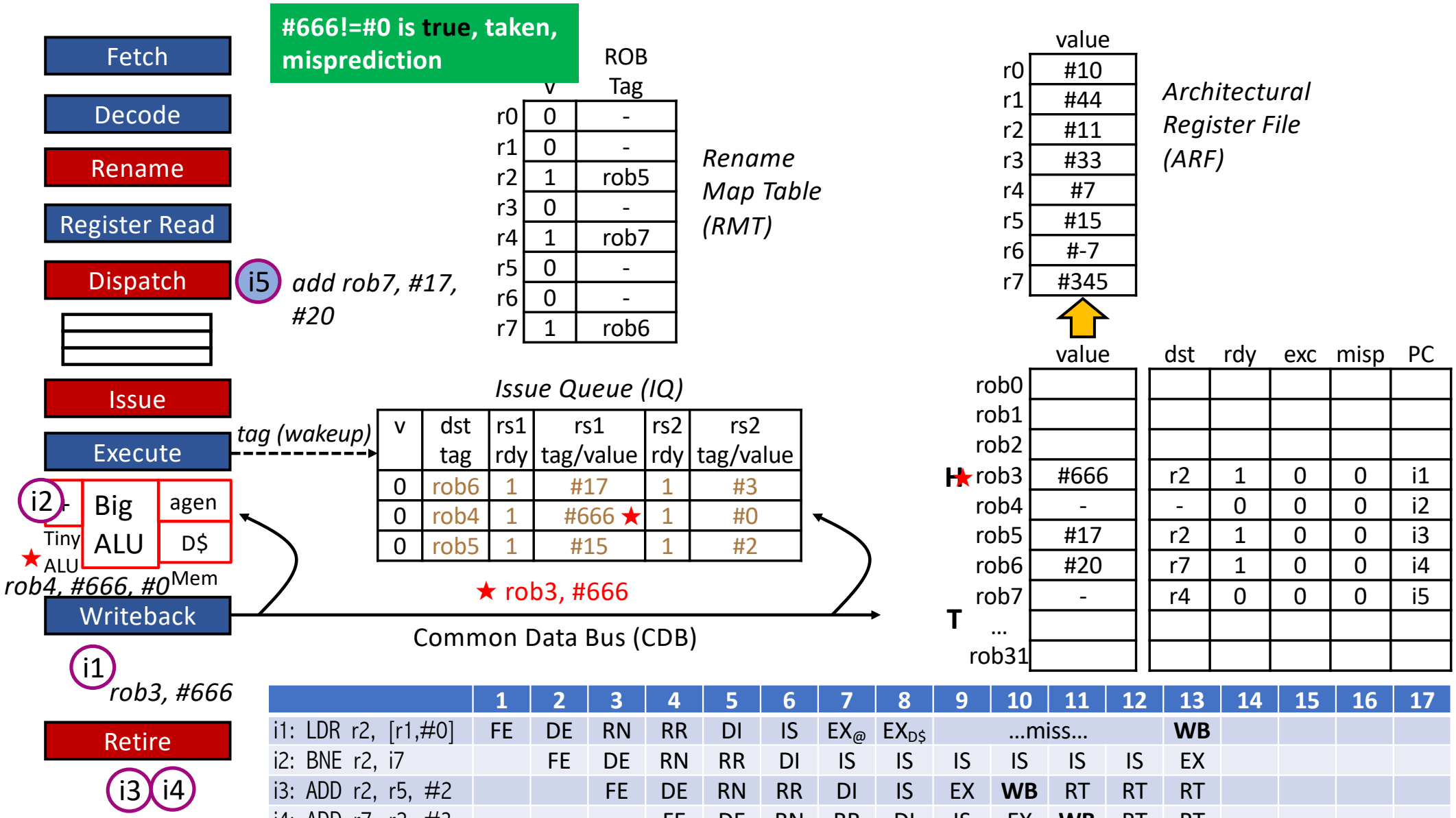
- **Compared to scoreboard only**
  - ROB did not degrade performance
    - Fetch did not stall as before (tolerated D$ miss)
  - In-order retirement did not impede OOO, speculative execution

- **Recovery**
  - ROB was not called upon for recovery
  - Only leverages ROB for renaming
  - We can see the danger of misprediction without ROB

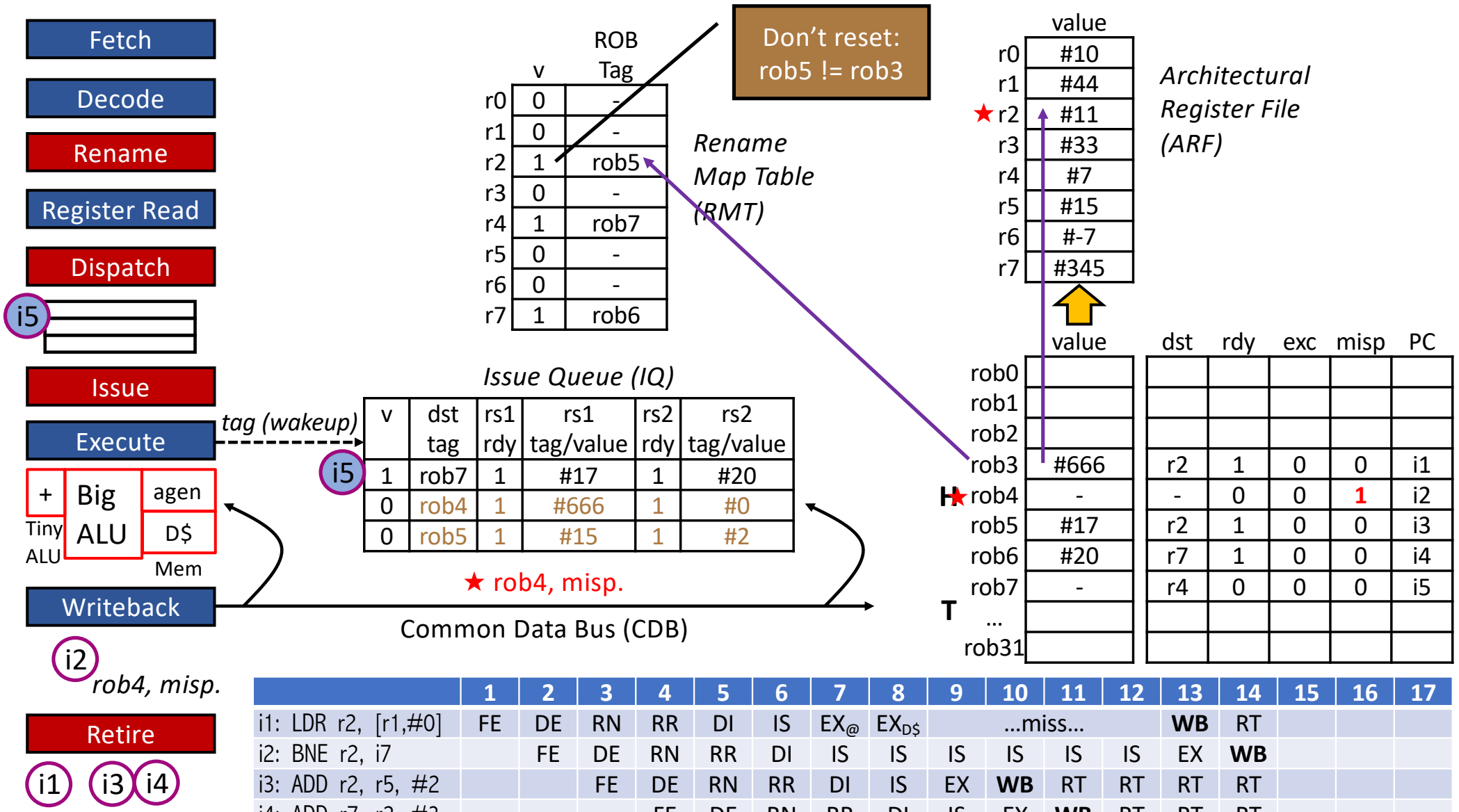
Recovery

Revise the previous scenario assuming **mispredicted** branch

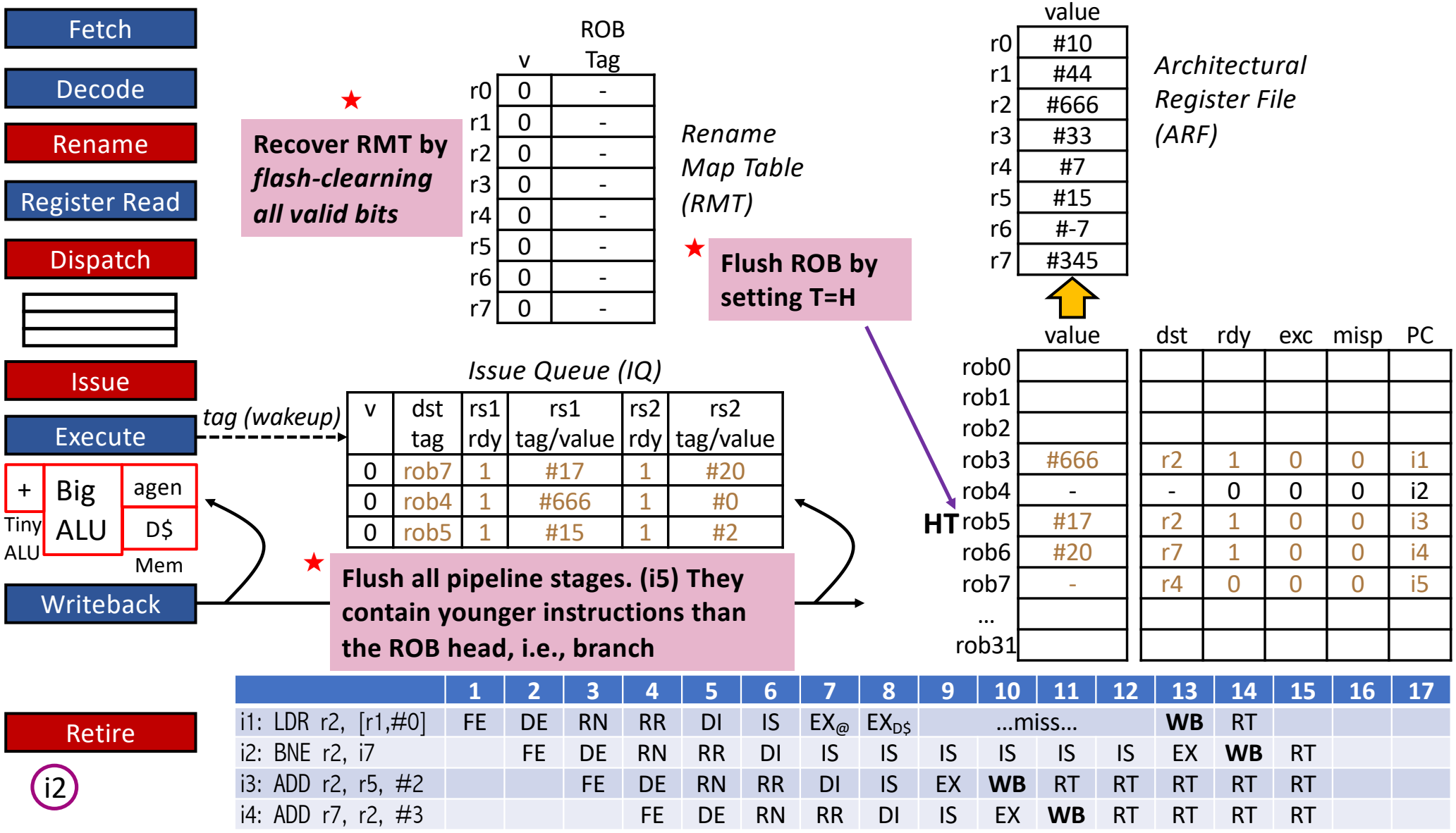
- i1 (load instruction) gets the value #666 instead of #0



	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**				
i2: BNE r2, i7		FE	DE	RN	RR	DI	IS	IS	IS	IS	IS	IS	EX				
i3: ADD r2, r5, #2			FE	DE	RN	RR	DI	IS	EX	**WB**	RT	RT	RT				
i4: ADD r7, r2, #3				FE	DE	RN	RR	DI	IS	EX	**WB**	RT	RT				

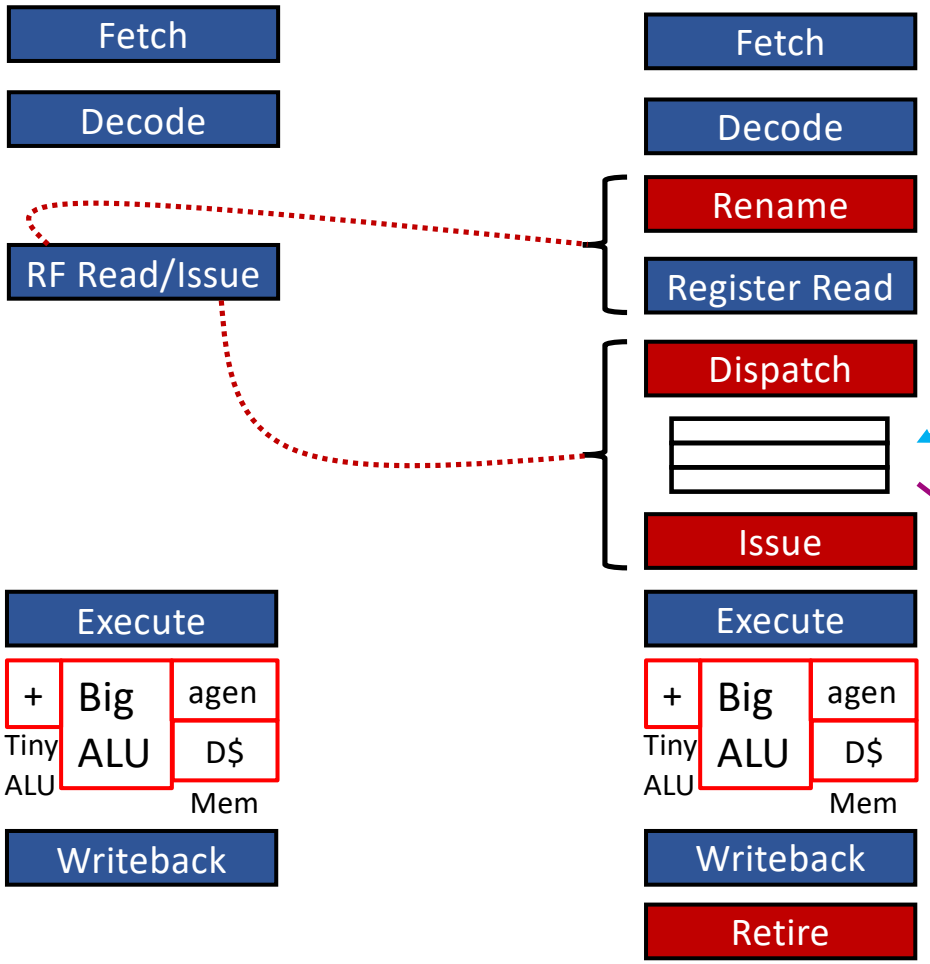


	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
i1: LDR r2, [r1,#0]	FE	DE	RN	RR	DI	IS	$EX_{@}$	$EX_{D\$}$	...miss...				**WB**	RT			
i2: BNE r2, i7		FE	DE	RN	RR	DI	IS	IS	IS	IS	IS	IS	EX	**WB**			
i3: ADD r2, r5, #2			FE	DE	RN	RR	DI	IS	EX	**WB**	RT	RT	RT	RT			
i4: ADD r7, r2, #3				FE	DE	RN	RR	DI	IS	EX	**WB**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



	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: BNE 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



Expanded Register File:

ARF (committed state) + ROB (speculative state)

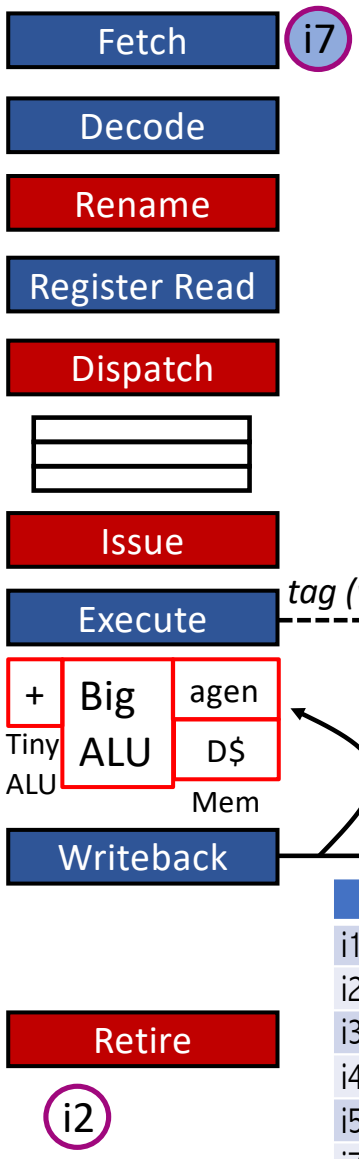
Provides for recovery and eliminated WAR/WAW

Insert instructions in order

Issue Queue (IQ)

Remove instructions out of order

Commit instructions in-order from ROB to ARF



tag (wakeup)

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	#666	1	#0
0	rob5	1	#15	1	#2

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

Architectural Register File (ARF)

	value	dst	rdy	exc	misp	PC
rob0						
rob1						
rob2						
rob3	#666	r2	1	0	0	i1
rob4	-	-	0	0	0	i2
HT rob5	#17	r2	1	0	0	i3
rob6	#20	r7	1	0	0	i4
rob7	-	r4	0	0	0	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: BNE 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		
i5:					*Can fetch more insts*				FE	DE	RN	RR	DI	IS	EX		
i7:																FE	

OOO Execution of Loads and Stores

- Loads and stores also execute out of order
- A load and a store can be done safely out of order, provided they access different addresses
- If a load and store access the same address, then the reordering could result in a hazard
  - Load must not consume data from a younger store (WAR respected)
  - Most recent store behind (i.e., older) a younger load must produce value for it (RAW respected)

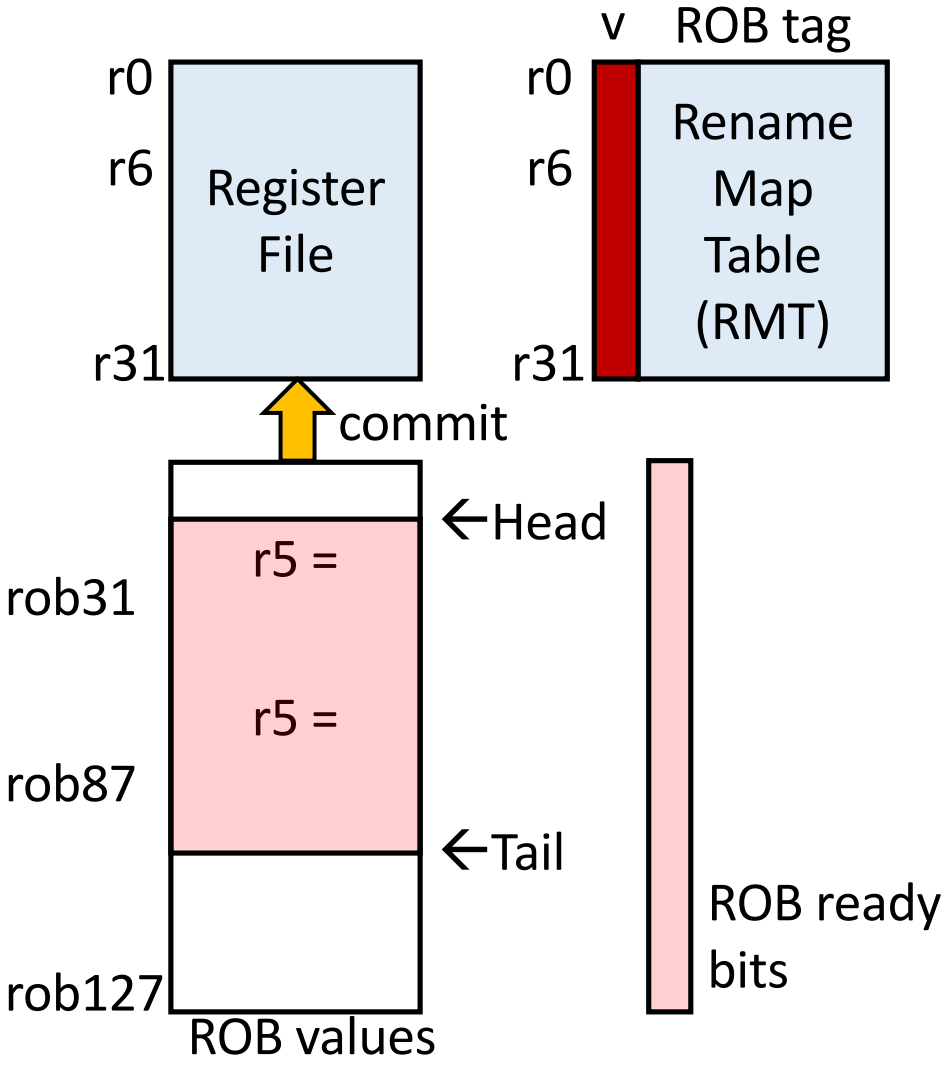
Speculative Load Execution

- Load executes speculatively, i.e., before the addresses of all **older** stores are known (using a structure called load-store queue or LSQ)
- Load searches for all speculative (older) stores with matching addresses
  - Load gets the "best it can get" (cache, main memory, ROB, RF, ...)
  - If it gets a stale value, recovery mechanism will save the day
- On execution, store cancels all speculative (younger) loads with matching addresses
- In fact, once we have speculation support, we can predict other things
  - Speculating on register values (value prediction)!

IBM 360/91 Floating Point Unit

- ARF+ROB is based on Tomasulo's Algorithm (1967)
  - Execute multiple floating-point instructions concurrently
  - The original machine was imprecise (no ROB) and used issue queue for renaming
  - It used "stall on branch," hence exploited 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

- Dynamic branch prediction
  - Choosing which instructions to execute next
- Speculation
  - Allowing the execution of instructions before dependences are resolved (with the ability to undo the effects of an incorrectly **speculated** sequence)
- Dynamic scheduling
  - Dealing with the scheudling of different combinations of instructions
  - Dynamic scheduling without speculation exploits limited ILP as branches must resolve prior to actually executing instructions in the predicted path
- Register renaming
  - Rename logical/arch. registers to an extended set of physical registers (avoid WAR/WAW)
- Hardware-based speculation
  - Dynamic branch prediction + dynamic scheduling + speculation
  - Renaming is an **optional** but **important** optimization
- Precise interrupts
  - On an exception, the architectural state must correspond to the sequential execution

Modern Processor Design: Key to High Performance

- Eliminate false dependences with register renaming
- Use dynamic branch prediction to "speculatively" fetch instructions and fill the issue queue with many instructions
- Issue instructions out of order to keep all functional units busy
- Use superscalar to fetch, decode, ..., issue "many instructions" each cycle
- **Key reason for above:** Hide memory latency. Work underneath a cache miss
  - Waiting for memory **KILLS** performance. Memory accesses are common

Modern Processor Design: Key to High Performance

- **Remember:** Not much ILP in a small instruction sequence (too many dependences)
- **Remember:** Need a large scheduling window (aka lots of slots in the issue queue) to find independent instructions (with properly sized ROB)
- Modern processors use a variation on **ARF+ROB** approach
- MUST understand prior designs to build new, better, ones for emerging applications

A Historical 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, load misses
    - Power, energy, and security issues (no time to cover here)
  - 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: Three Classes

- **Statically scheduled superscalar processor**
  - Compiler reorganizes (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**
  - **Conceptually same as above:** Compiler does static scheduling
  - Compiler creates instruction words called bundles with up to 28 instructions in a bundle (instructions in a bundle are all independent of each other)
  - Compiler does **"very smart and deep" program** analysis to construct "good instruction schedules with high ILP"
  - No dependency checking in hardware

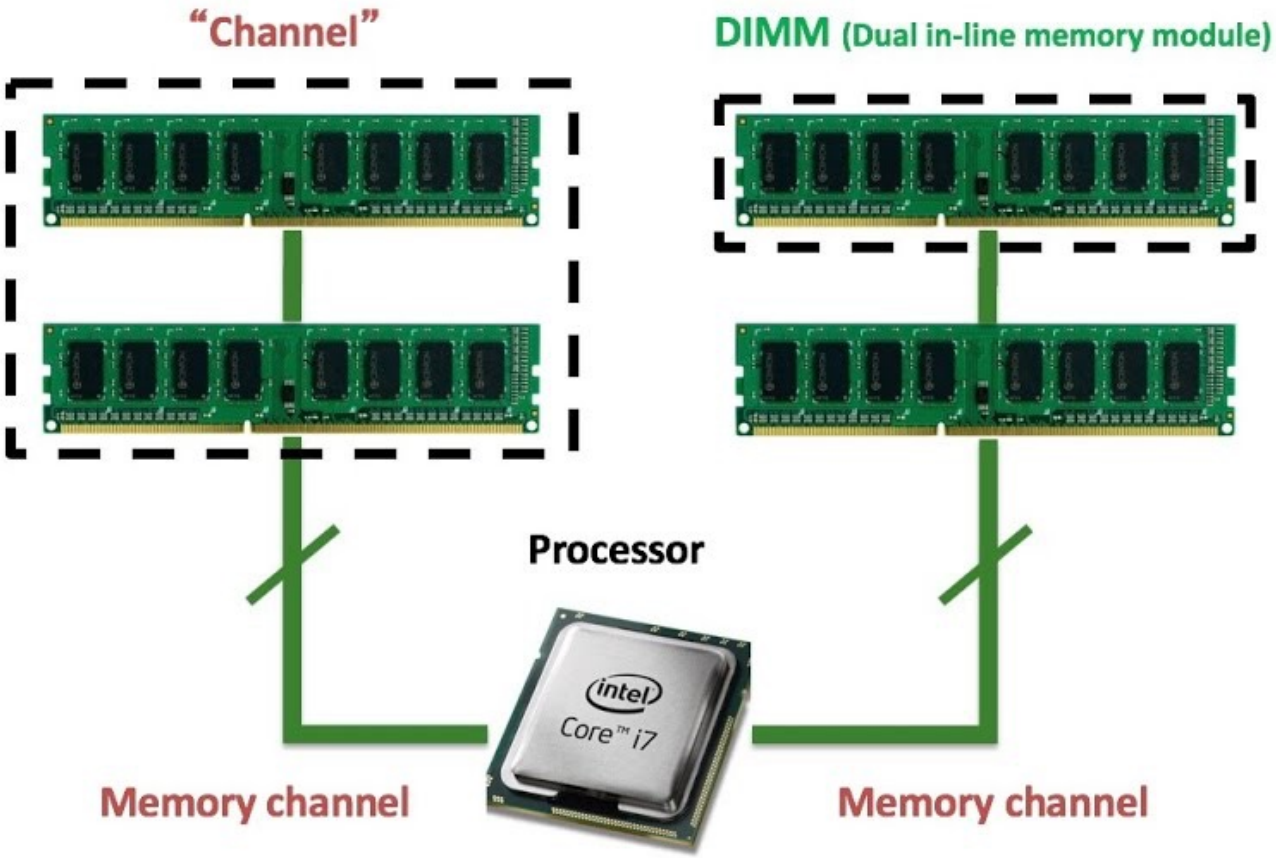
ILP Exploitation: Three Classes

- **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 instructions

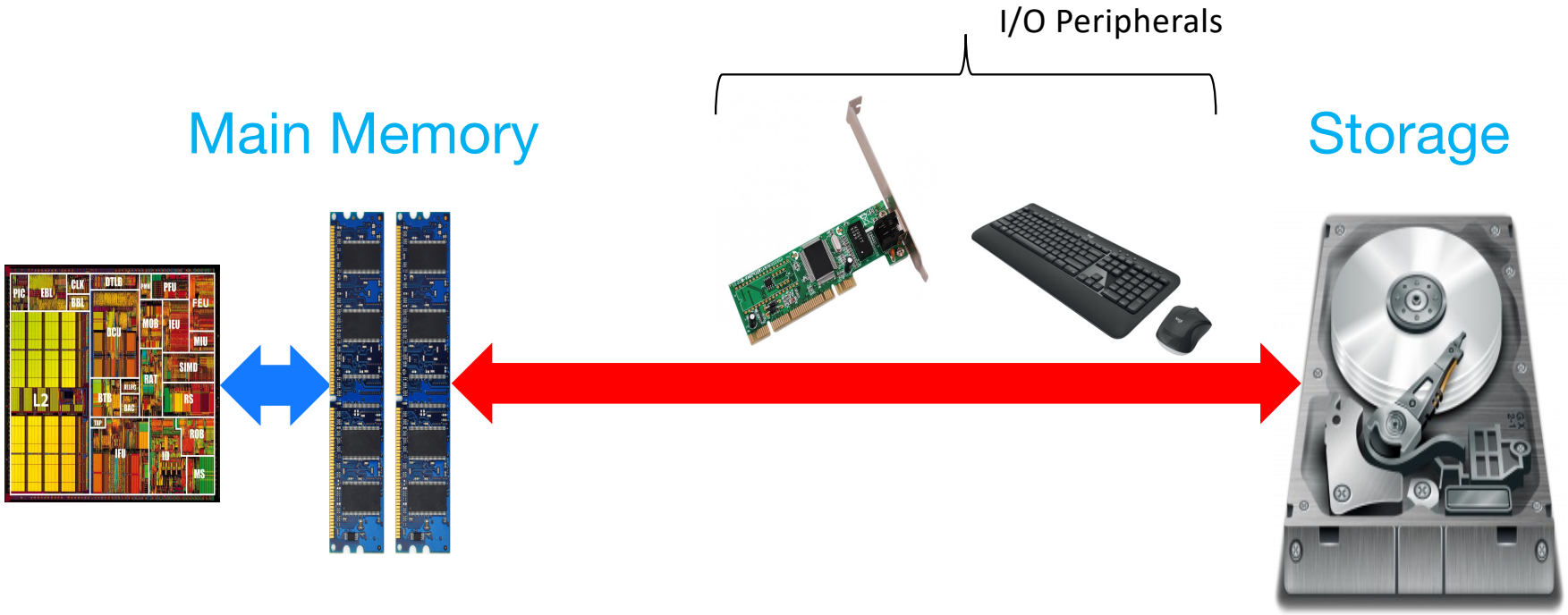
Memory Level Parallelism (MLP)

- OOO CPUs generate many load misses by exploiting ILP and the rest of the system must cooperate
- Caches are lock-up free (non-blocking)
- DRAM main memory can handle many requests in parallel using multiple channels, ranks, and banks
- Storage devices (SSDs) also have internal parallelism to support concurrent accesses

DRAM Subsystem



Parallelism across the system



What Else? Other Forms of Parallelism

- **Multicores (spatial parallelism, frequency increase not possible)**
  - AMD has CPUs with 128 cores

- **Hyperthreading (2-way hyperthreading, better utilize superscalarOOO)**
  - Difficult to find N (4, 8, or 12) instruction to issue for execution every cycle
  - Multiple software threads share resources of a superscalar OOO core (separate RF + shared OOO machinery)

- **SIMD registers (single instruction, multiple data)**
  - Wide (up to 1024-byte) registers to execute same instruction on multiple items

What Else? Security Vulnerabilities

- Hardware speculation leads to security vulnerabilities

- Why do we teach OOO?
  - Performance, energy, security (increasingly more imp.)

- We do not have time to cover this subject in this course

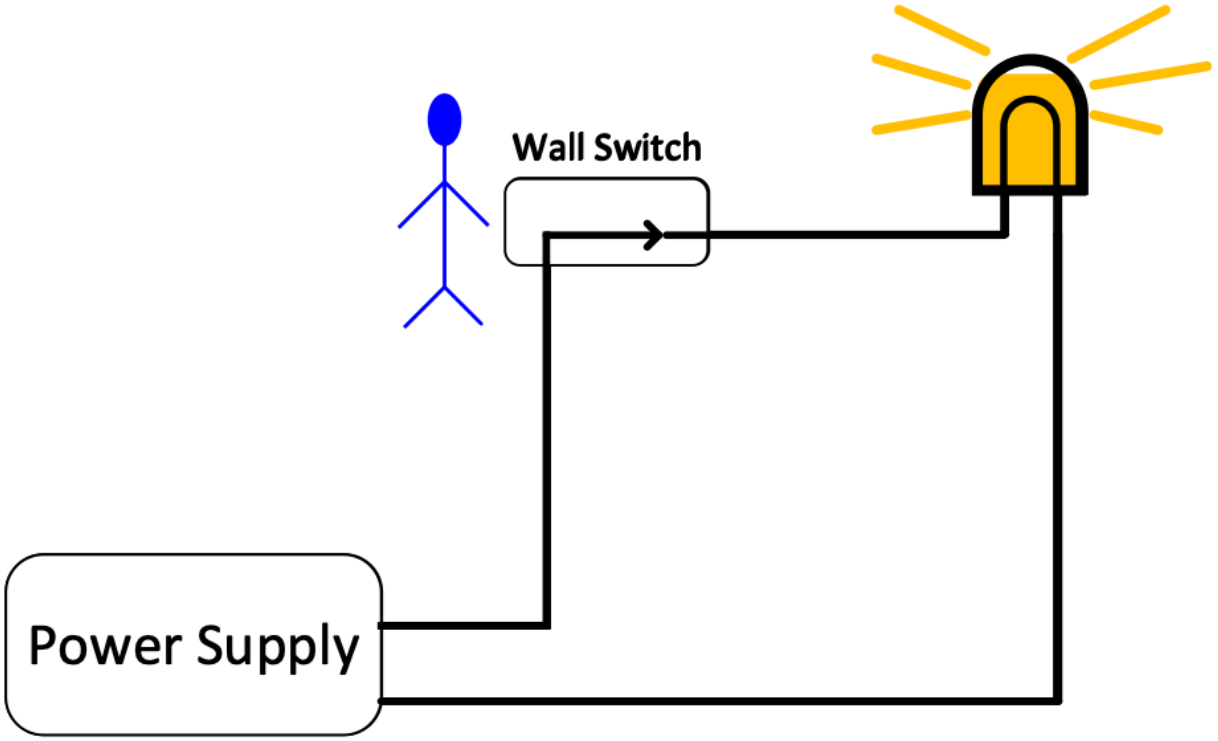
- **Homework**
  - **AI Prompt:** Discuss how hardware speculation enables Spectre variant 1

What Else? Security Vulnerabilities

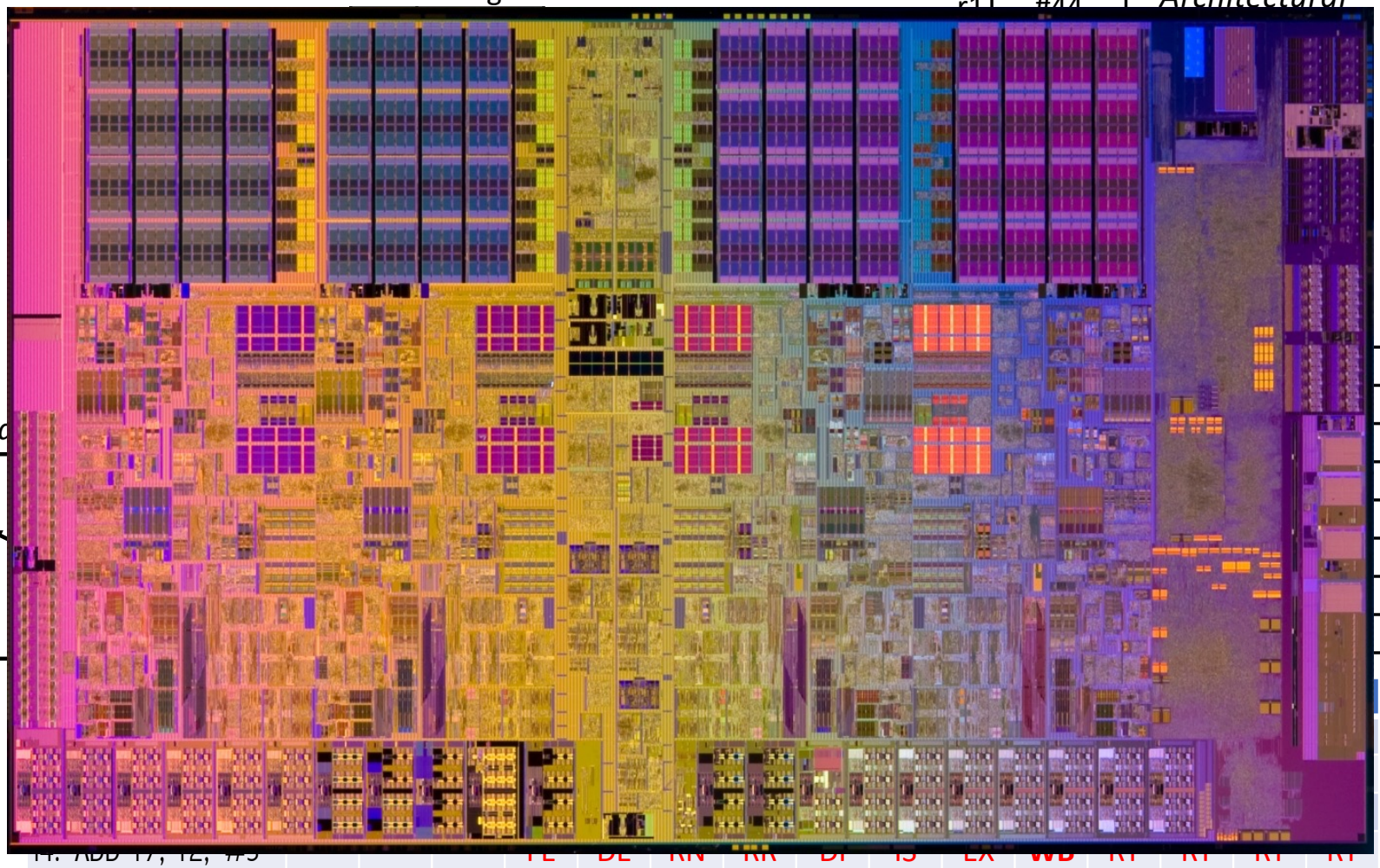
- Series of guest lectures on **vulnerability research** from industry practitioners as part of COM3703 Software Security

- **Venue: Psychology Building 39, Room G06**
- Monday, 19th of May, 11am-12pm: **Cameron Jack**, Director of Product Strategy at InfoSect. "*What does a Vulnerability Research career look like, and is it for me? Also a pwn2own bug walkthrough*"
- Tuesday, 20th of May, 10am-11am: **Angus Atkinson**, Vulnerability Researcher at InfoSect (and ANU alumni). "*Walkthrough of an N-day Android GPU driver vulnerability*"
- Wednesday, 21st of May, 10am – 11am: **Daniel Wood**, Senior Vulnerability Researcher at Interrupt Labs. "*Vulnerability Research: From Userland to Kernel Space*"

From Here



To Here



Fetch
Decode
Rename
Register Read
Dispatch
Issue
Execute
Writeback
Retire
i7
i2
+
Tiny ALU
Big ALU
agen
D$
Mem
ROB
v
Tag
value
r0
#10
Architectural
misp
PC
0 i1
0 i2
0 i3
0 i4
0 i5
16
17
i5:
Can fetch more insts
FE DE RN RR DI IS EX
i7:
FE

The End!