Hardware Speculation: The Key to High Performance in Modern Processors

Shoaib Akram
shoaib.akram@anu.edu.au

Australian
National
University

Introduction

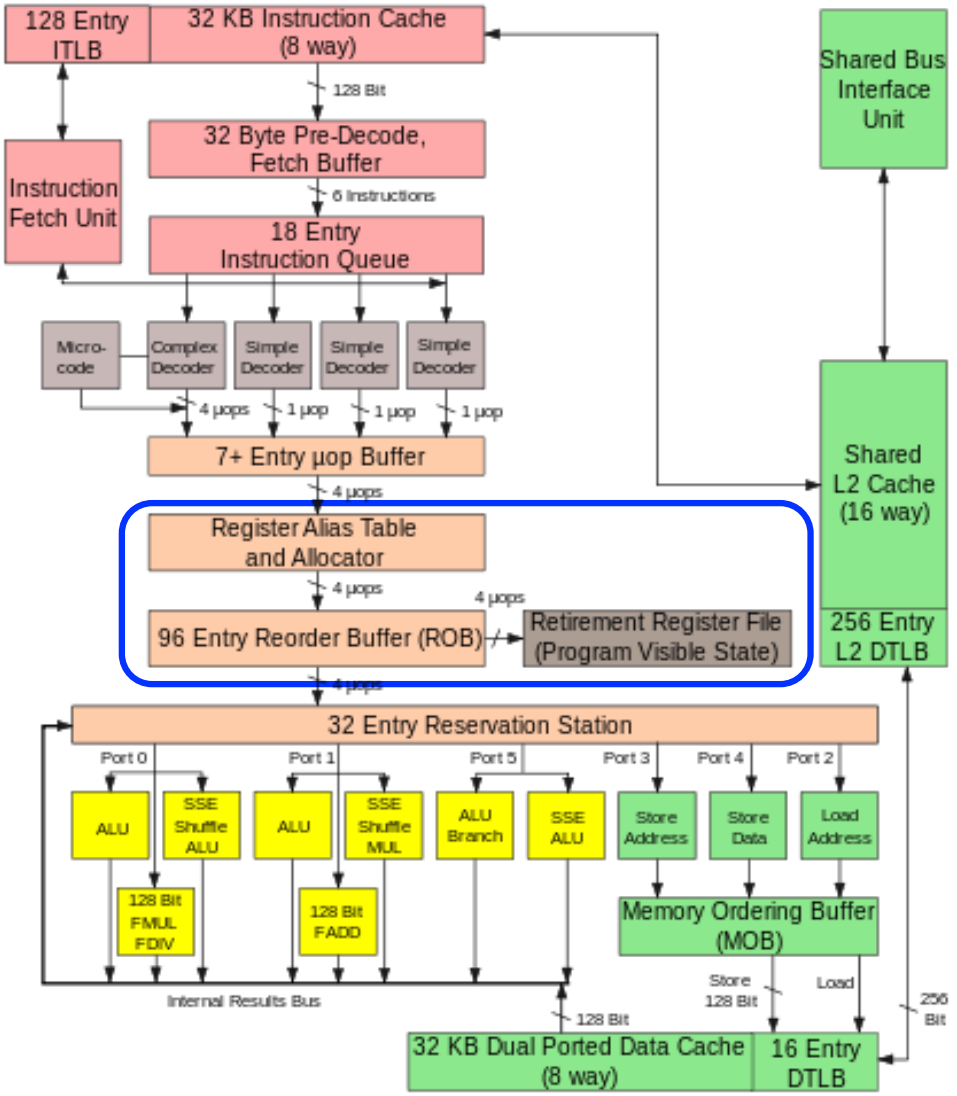
- Von Neumann model is inherently a sequential programming model

- High performance requires executing many instructions each clock cycle

- **Parallelism in modern machines**
  - Pipelining
  - Spatial duplication

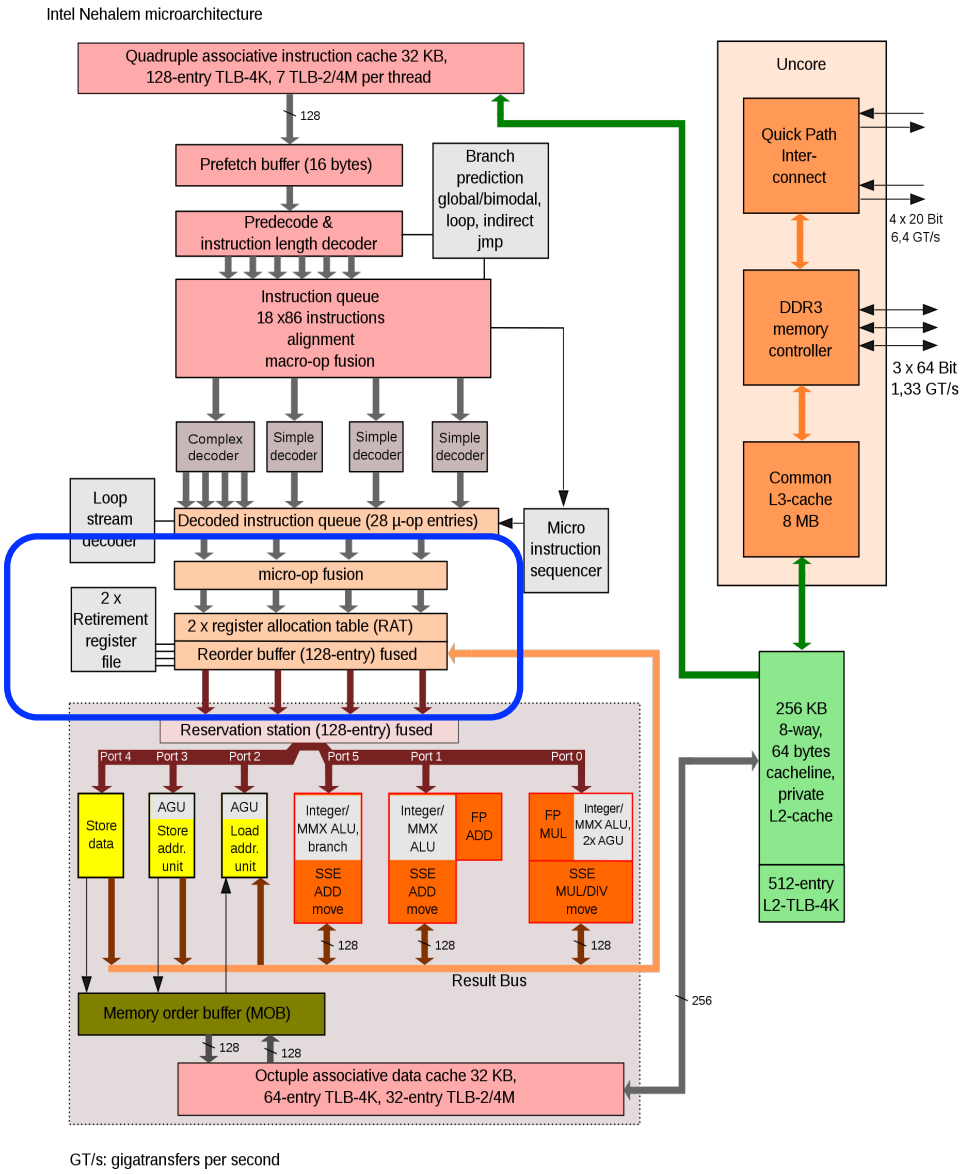
- Key problem: identifying independent instructions for concurrent execution

Intel Yonah
2006
Core 2

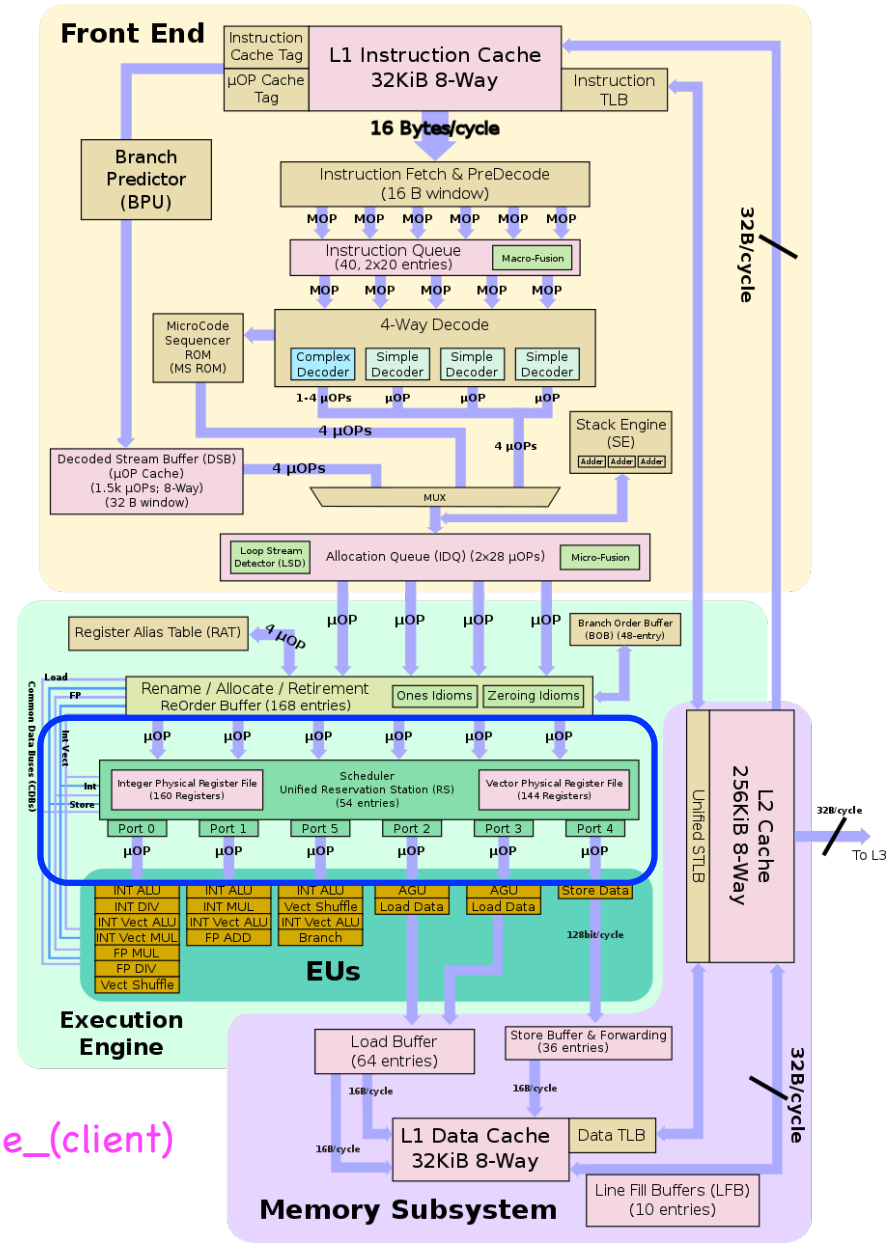


Intel Core 2 Architecture

Intel Nehalem 2008
Core i5, i7



Intel Sandy Bridge
2010
Core i5, i7



https://en.wikichip.org/wiki/intel/microarchitectures/sandy_bridge_(client)

Fundamentals

Von Neumann Model

Stored program

Sequential instruction processing

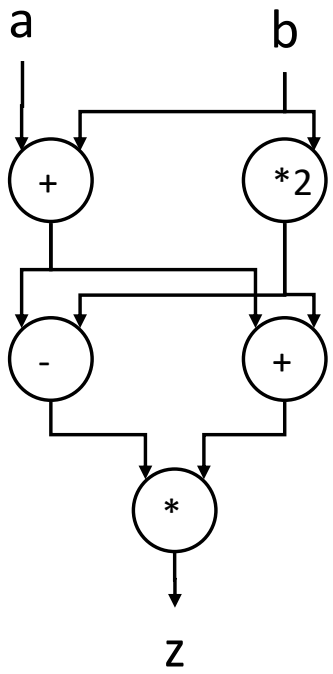
Von Neumann vs. Dataflow

- Consider a Von Neumann program
  - What is the significance of the program order?
  - What is the significance of the storage locations?

```
v = a + b;
w = b * 2;
x = v - w
y = v + w
z = x * y
```

Sequential

a, b are the only inputs
z is the only output

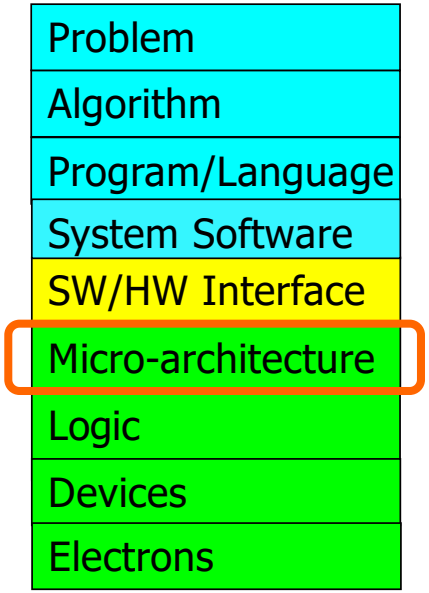


Dataflow

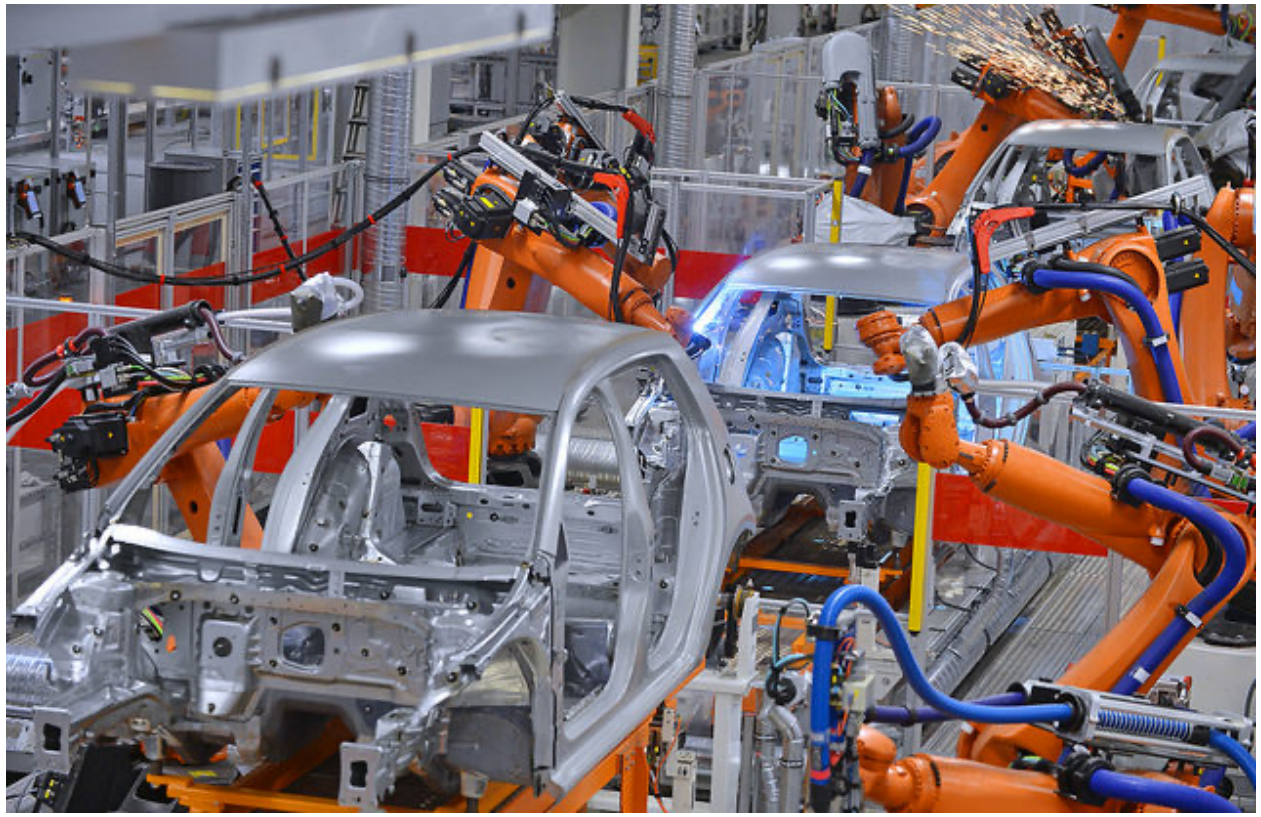
Which model is more natural to you as a programmer?

The von Neumann Model

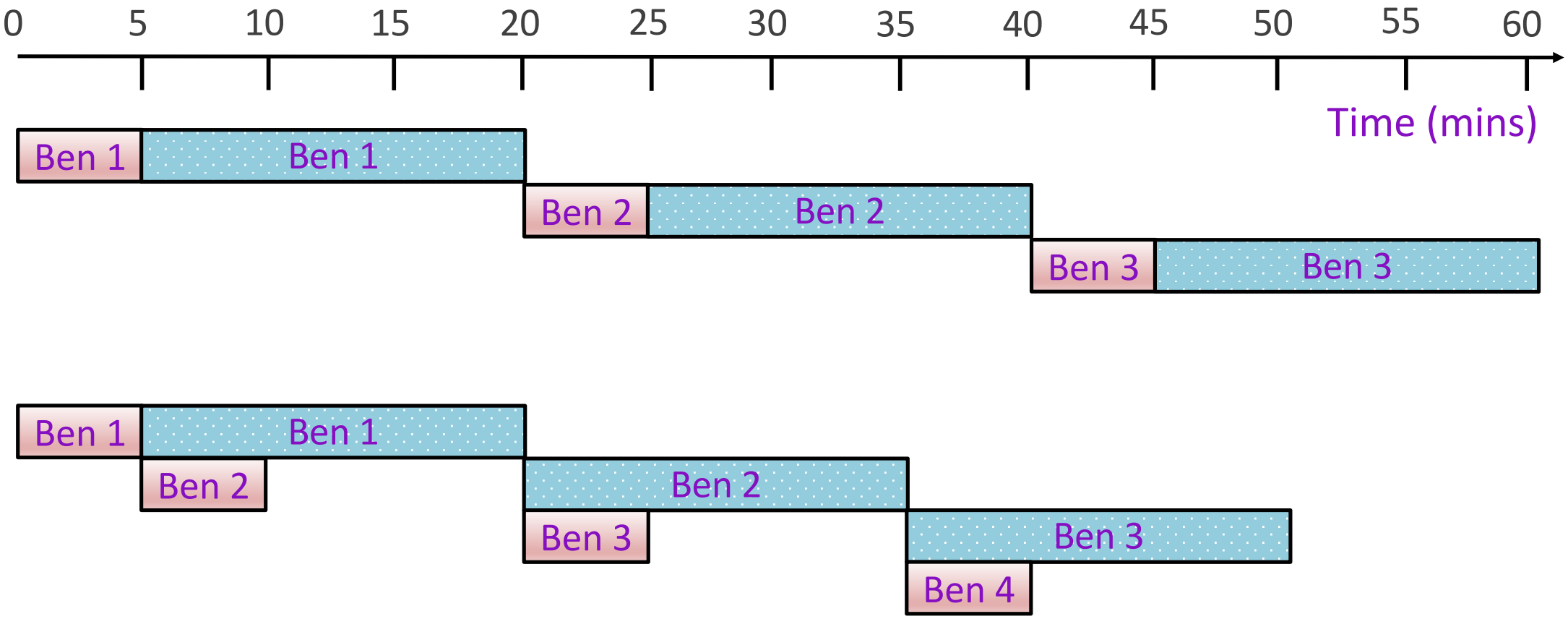
- All major *instruction set architectures* today use this model
  - x86, ARM, MIPS, SPARC, Alpha, POWER, RISC-V, …
- Underneath (at the microarchitecture level), the execution model of almost all *implementations (or, microarchitectures)* is very different
  - Pipelined instruction execution
  - Multiple instructions at a time
  - Out-of-order execution
  - Separate instruction and data caches
- But, what happens underneath that is ***not* consistent** with the von Neumann model is ***not* exposed** to software
  - Difference between ISA and microarchitecture



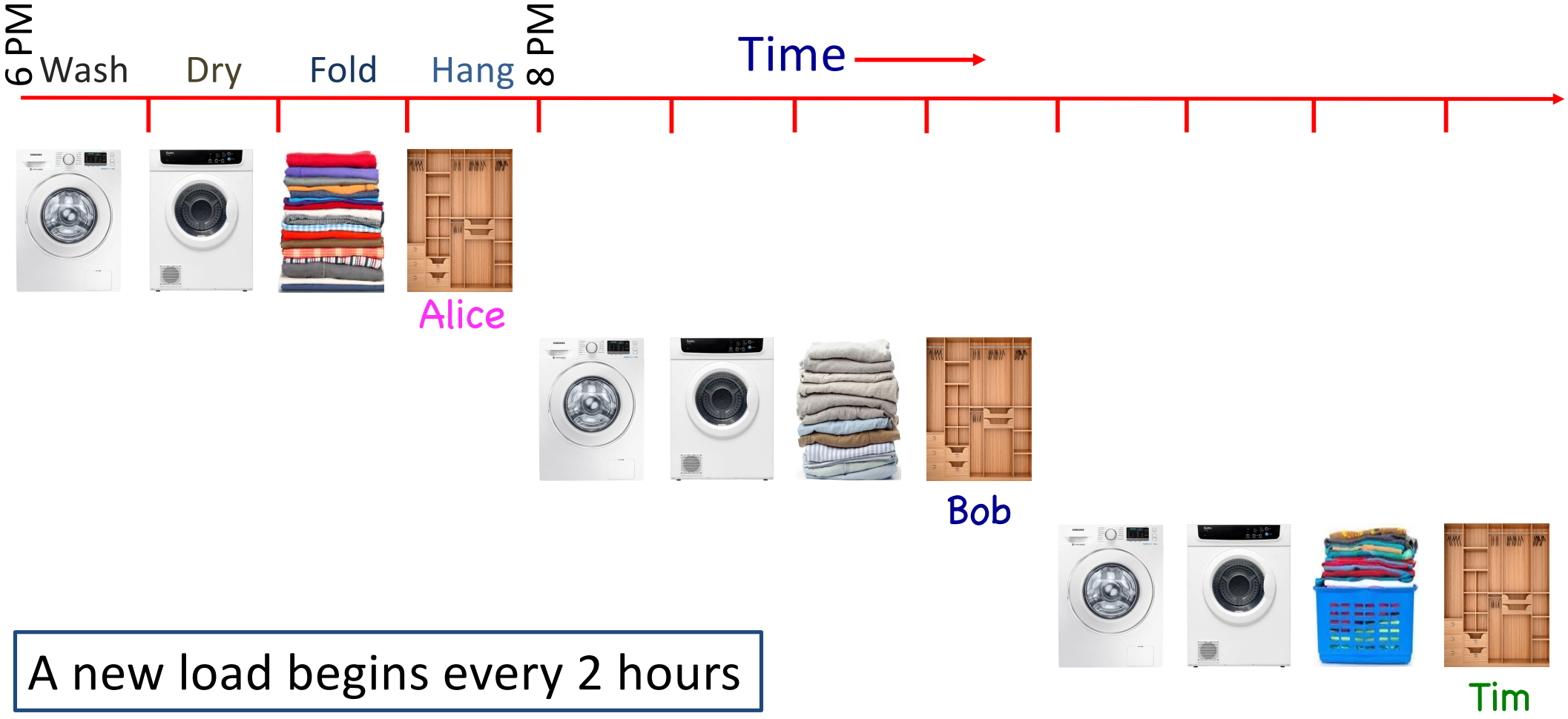
Automotive Pipeline



Cookie Pipeline

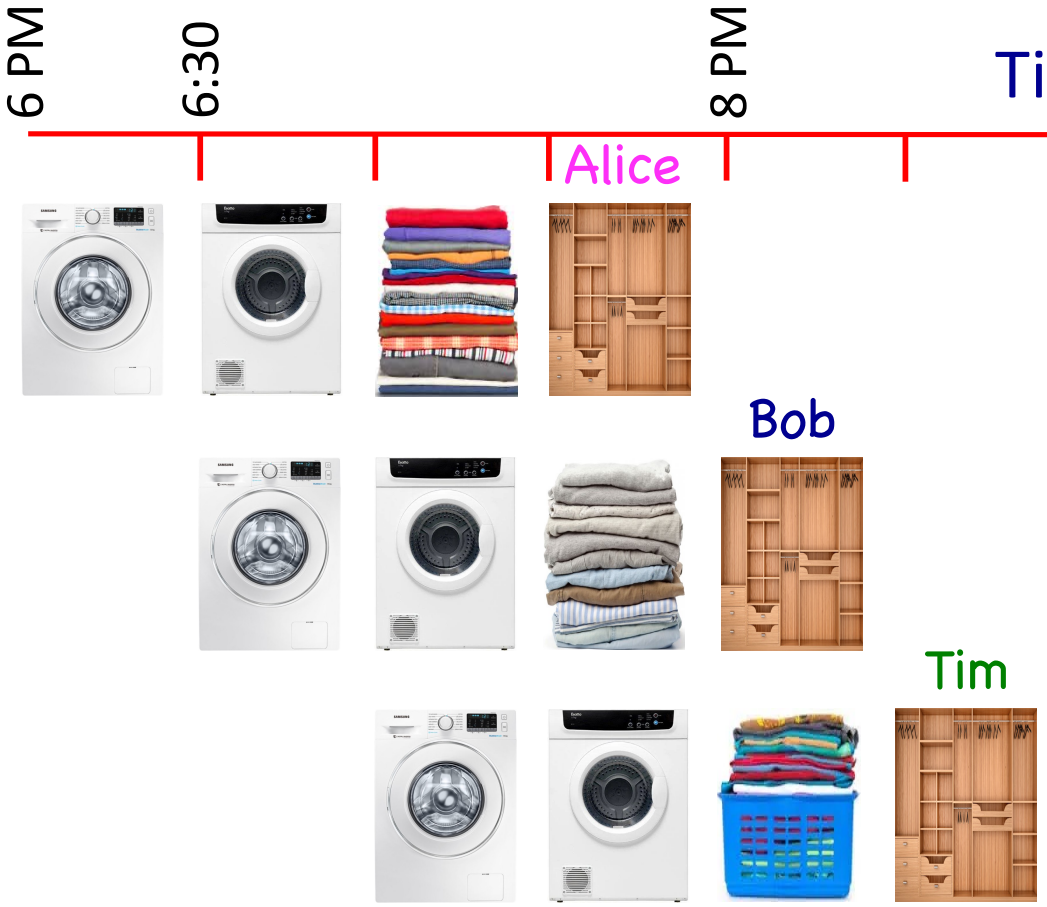


Sequential Laundry



A new load begins every 2 hours

Pipelined Laundry

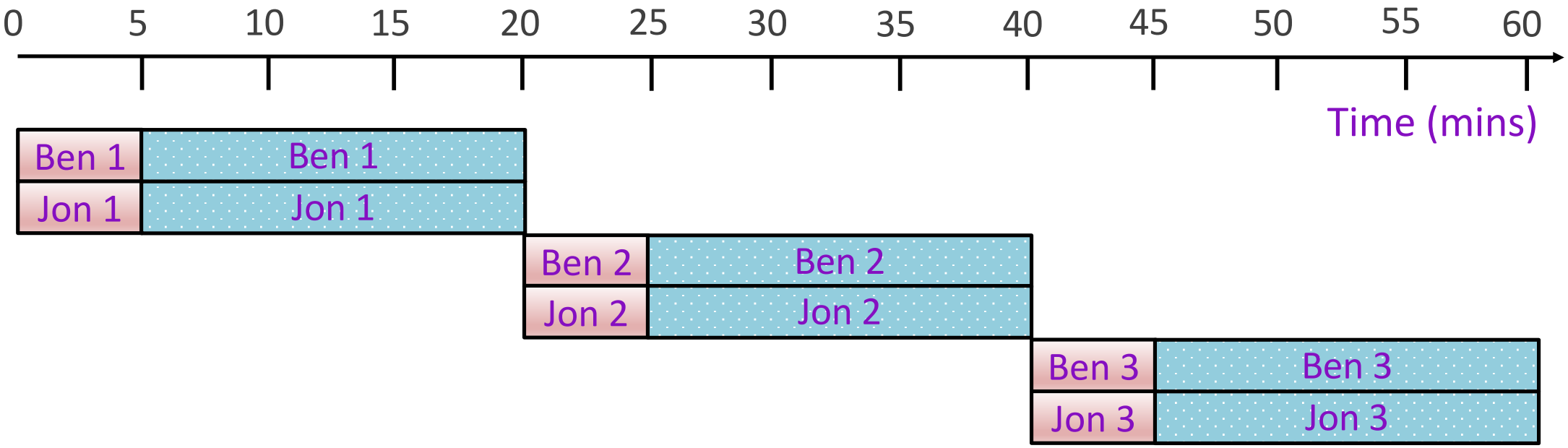


- A new load begins every 30 mins
  - 120 mins divided by 4
  - Speed-up of 4!

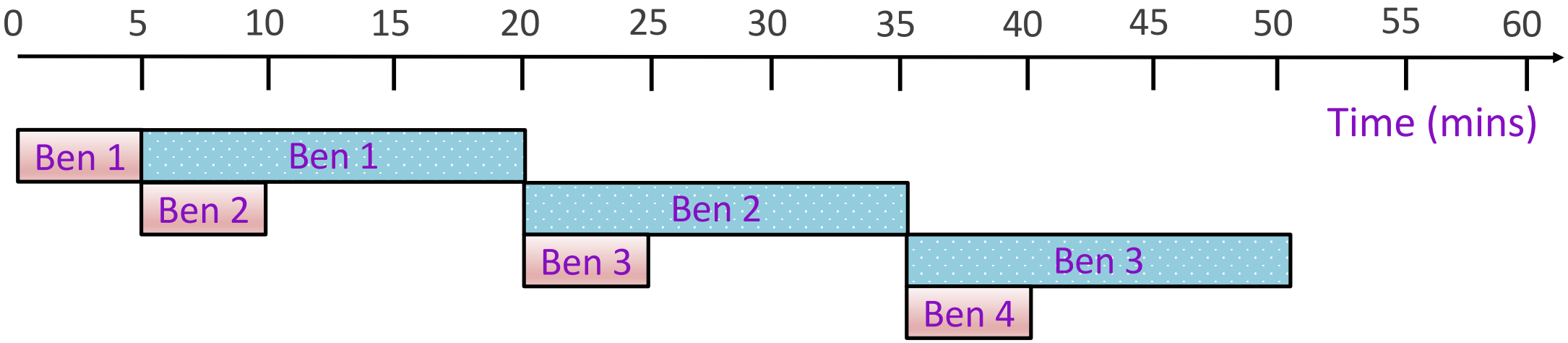
Recall: Cookie Parallelism

- Ben and Jon are making cookies. Let's study the latency and throughput of rolling and baking many cookie trays with
  - No parallelism
  - Spatial parallelism
  - Pipelining
  - Spatial parallelism + pipelining

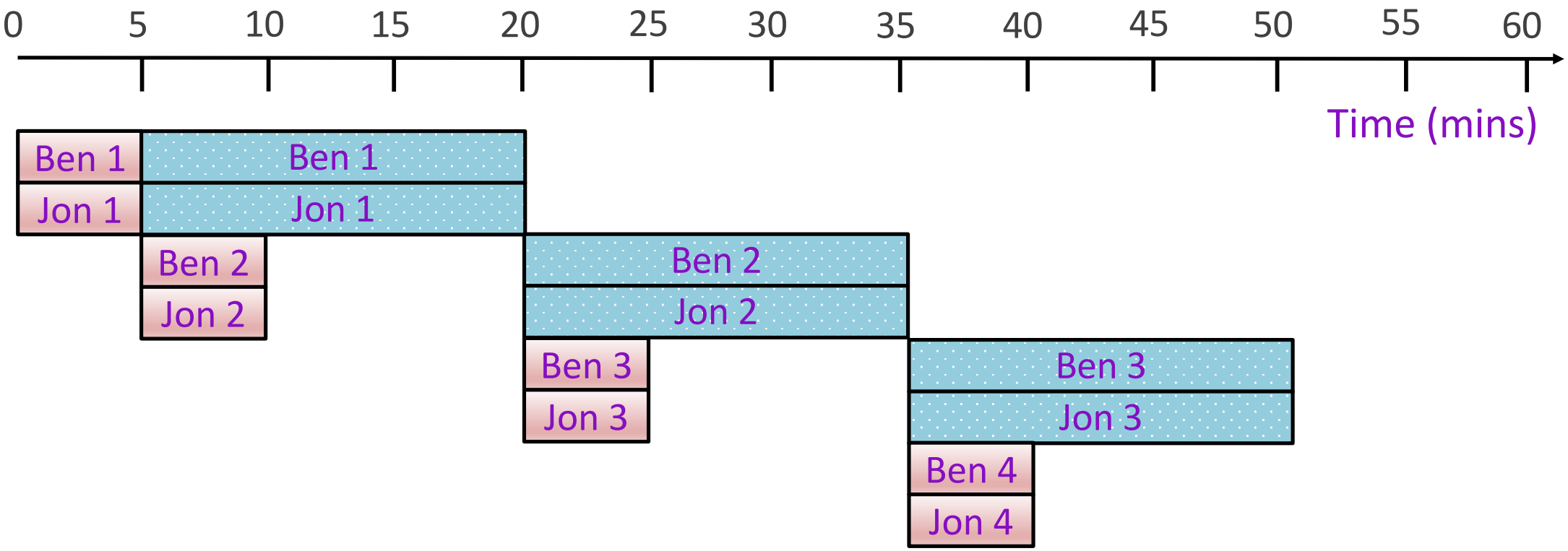
Spatial Parallelism (Ben & Jon)



Pipelining (Ben Only)



Spatial + Temporal Parallelism



Recall: Pipelining

- If a task of latency L is broken into N stages, and all stages are of equal length, then the throughput is N/L

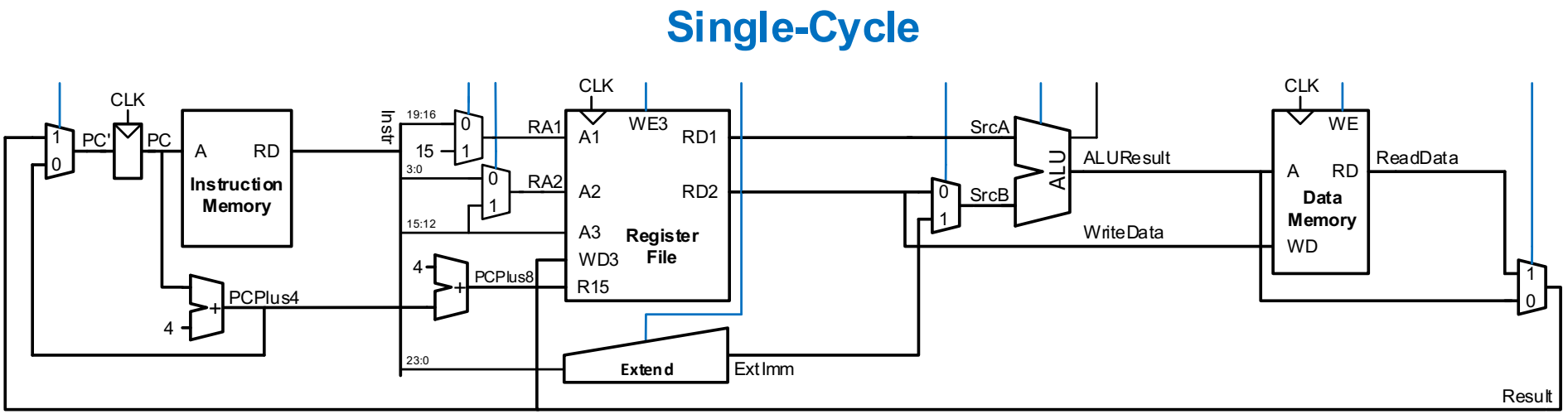
- The challenge of pipelining is to find stages of equal length

- Let's go back to baking cookies

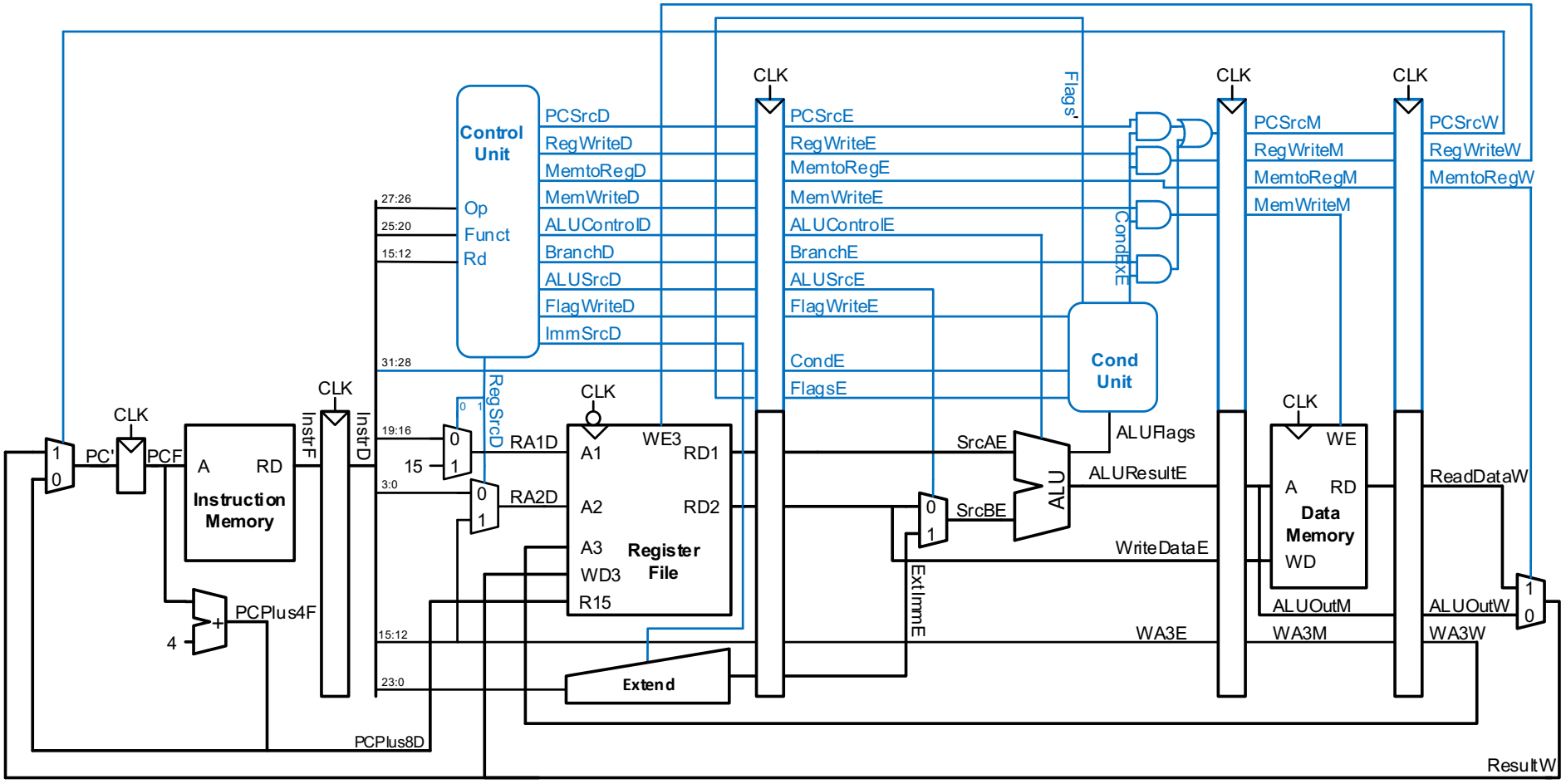
Recall: Pipelining Circuits

- Divide a large combinational circuit into shorter stages
- Insert registers between the stages
  - The outputs of one stage are copied into a register and communicated to the next stage
- Run the **pipelined** circuit at a **higher** clock frequency
  - Each clock cycle, data flows through the pipeline from left to the right
  - Multiple tasks can be spread across the pipeline

Single-Cycle Processor

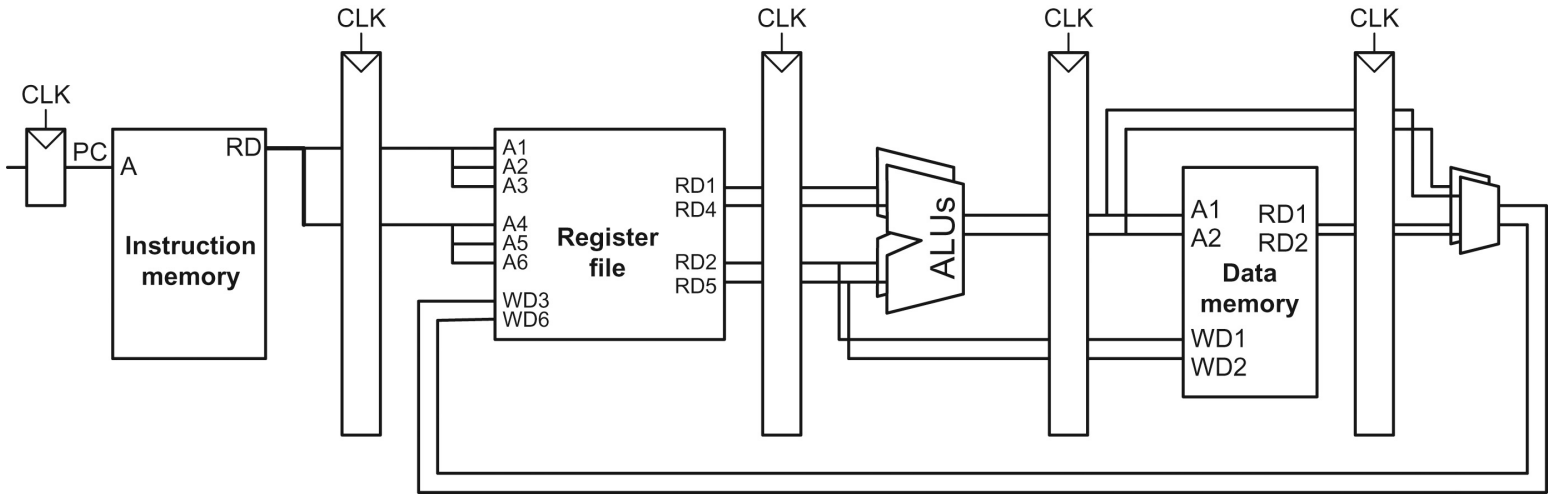


Pipelined Processor



Superscalar: Idea and Datapath

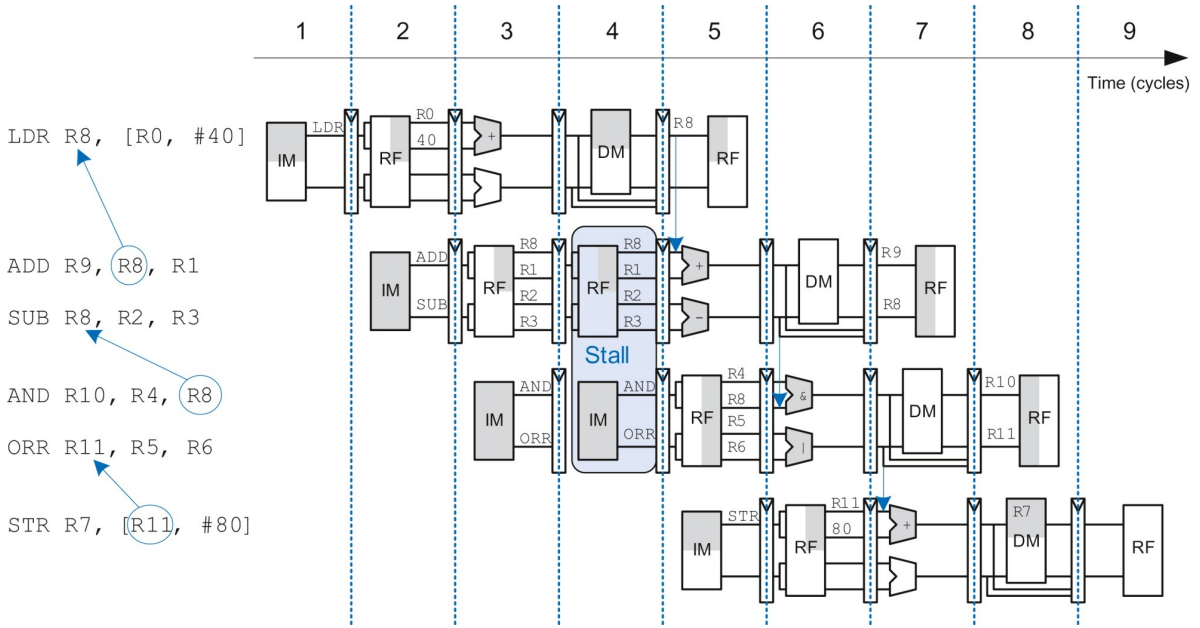
- Multiple copies of datapath hardware to execute instructions simultaneously
- **Example:** 2-way superscalar **fetches** and **executes** 2 instructions per cycle



- Requires 6-ported register file (4 reads, 2 writes), 2 ALUs, 2-ported data memory
- Ideal CPI = 0.5 and IPC = 2
- Dependencies and hazards inhibit ideal IPC
- Above figure does not show forwarding and hazard detection logic

Superscalar: Impact of Dependencies

- Example of program with data dependences

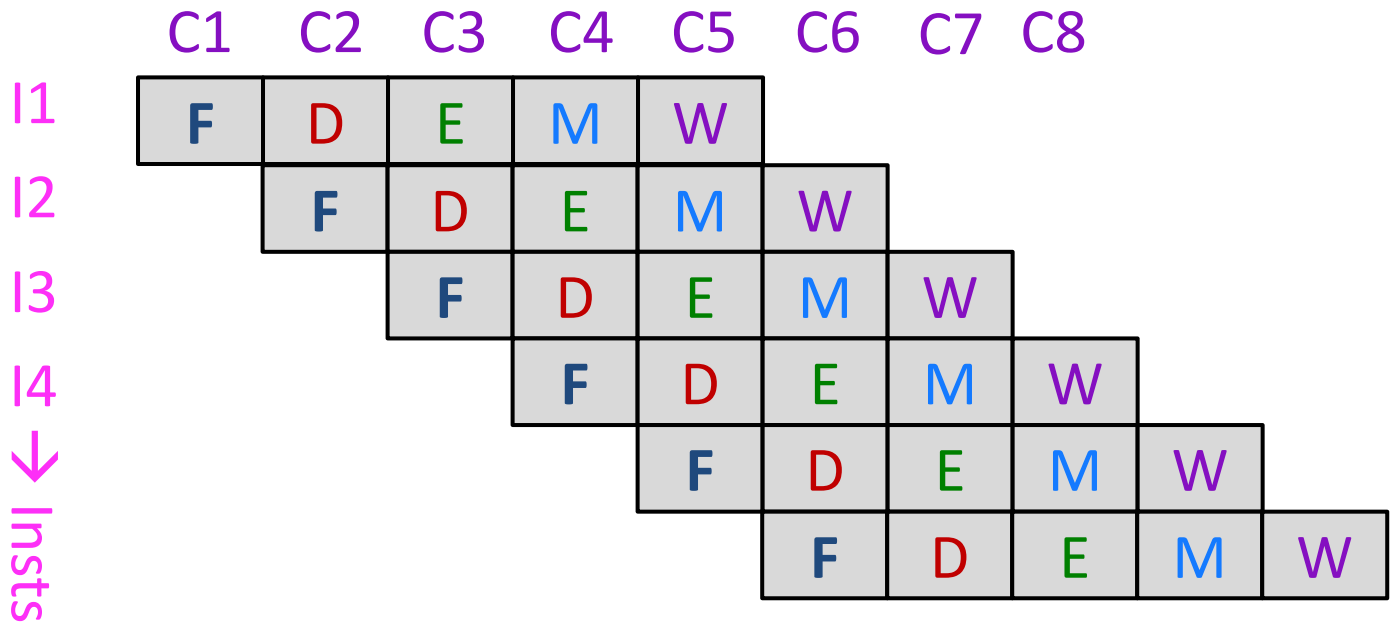


- The program requires 5 cycles to **issue** six instructions with an IPC of 1.2

Superscalar: Important Features

- Forwarding logic to steer results to ALU early (bypassing register file)
- Hazard detection logic to stall pipeline to respect true dependences
- Compiler can do "static scheduling" by analyzing code (simple machine)
  - Trace scheduling
  - Superblock, hyperblock
  - VLIW
  - Compiler can add fix-up code when scheduling past a basic block or have support from ISA
    - Instructions that trigger recovery from misspeculation

Pipelining: Simplified View

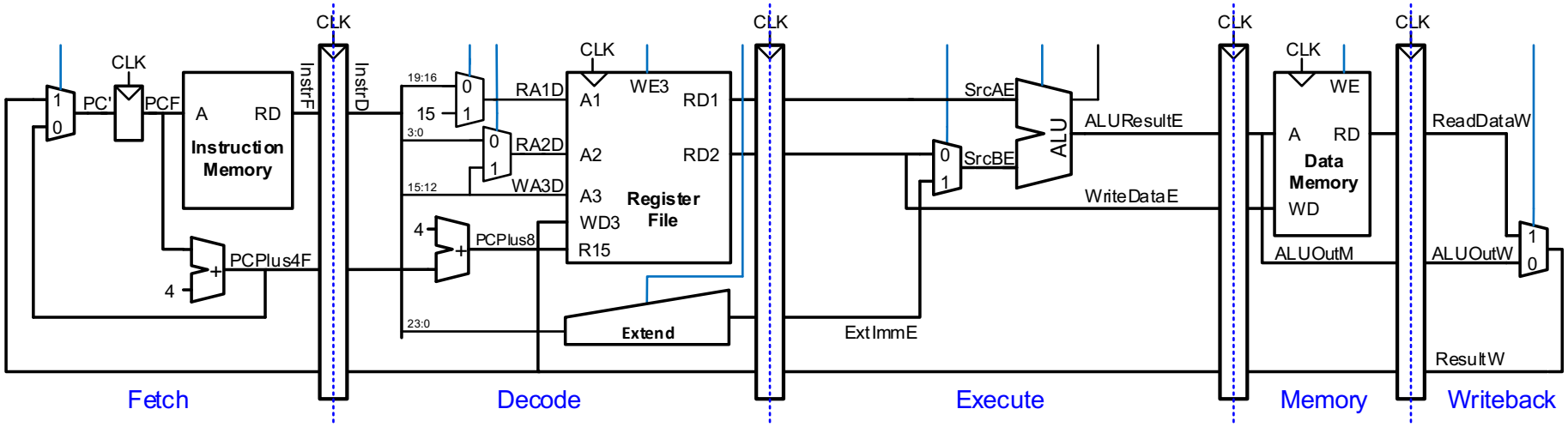


Pipeline Operation

- Consider the example instruction sequence

```
I1:   ADD   R0,   R5,   #10
I2:   ADD   R1,   R5,   #10
I3:   ADD   R2,   R5,   #10
I4:   STR   R0,   [R7, #4]
I5:   STR   R1,   [R7, #8]
I6:   STR   R2,   [R7, #12]
```

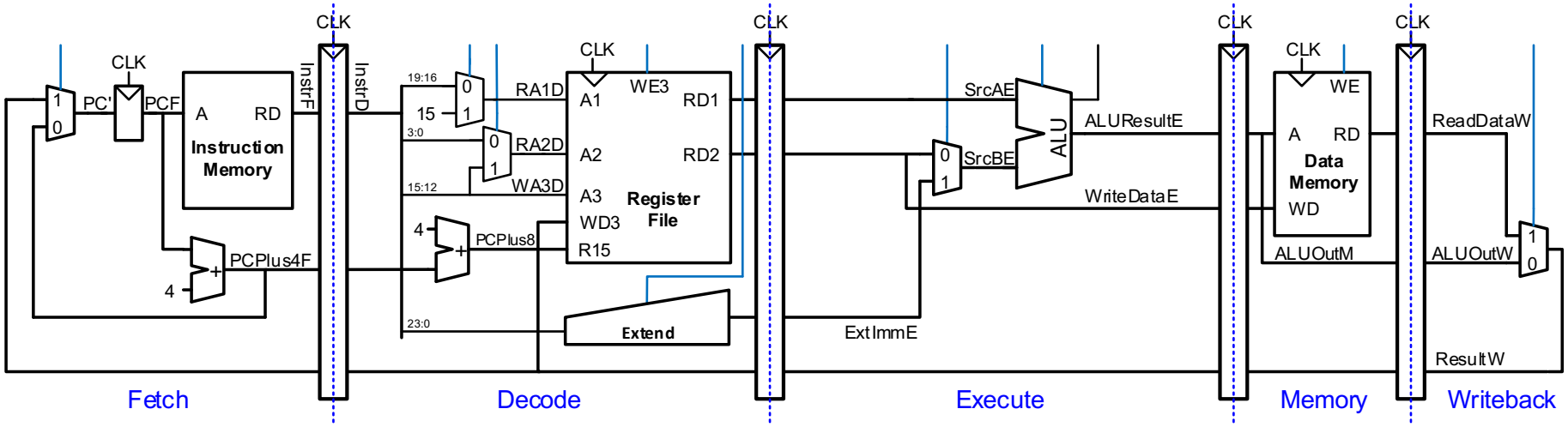
Pipeline Operation: Cycle 1



I1

- Is the pipeline fully utilized? NO

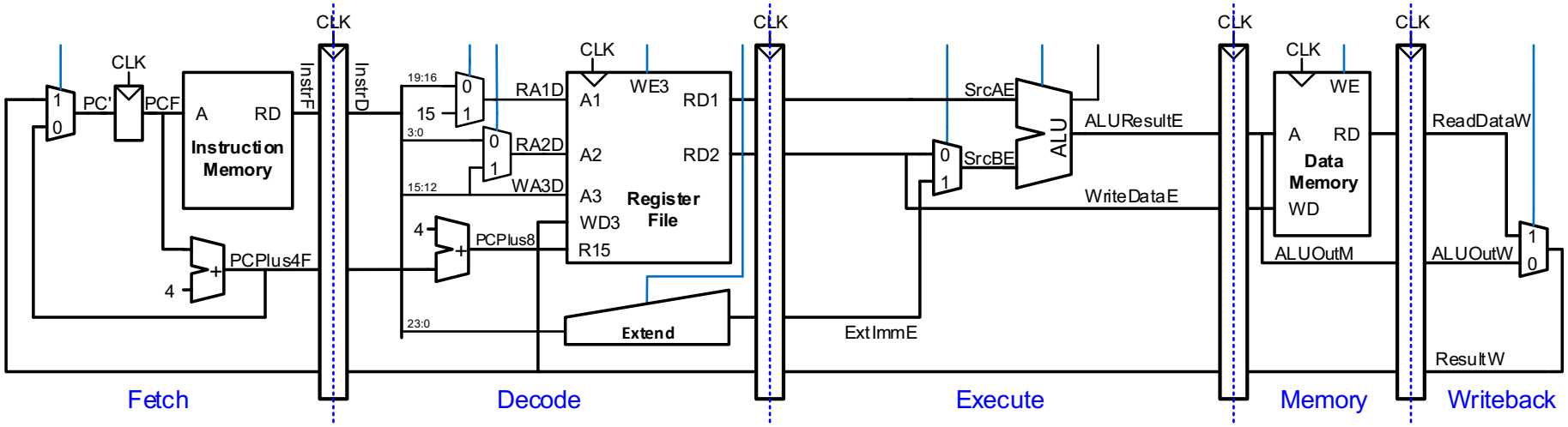
Pipeline Operation: Cycle 2



I2 I1

- Is the pipeline fully utilized? NO

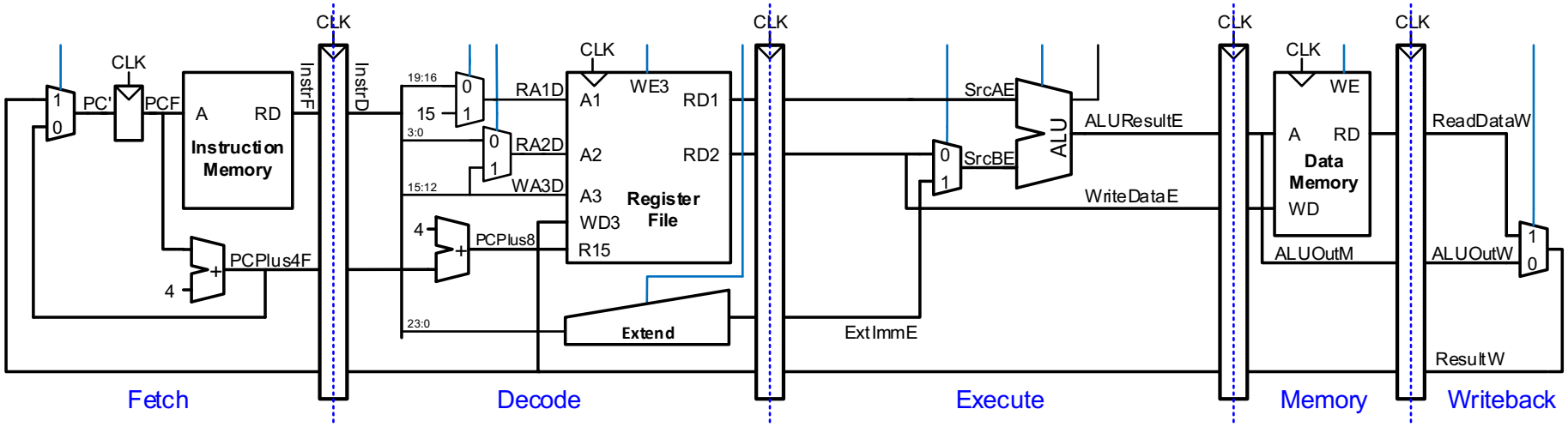
Pipeline Operation: Cycle 3



I3 I2 I1

- Is the pipeline fully utilized? NO

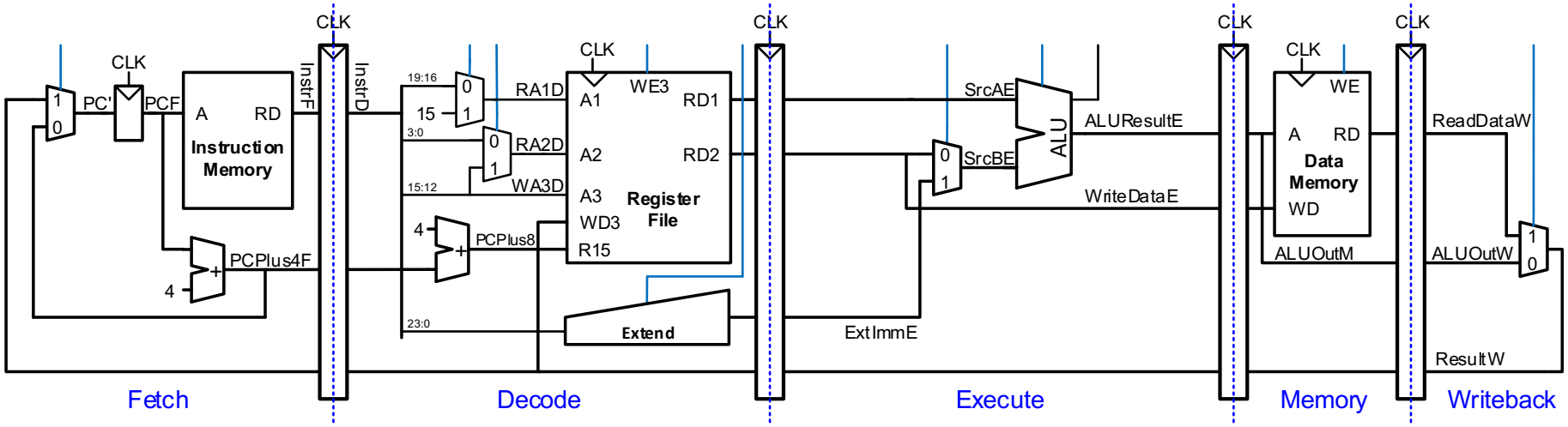
Pipeline Operation: Cycle 4



I4 I3 I2 I1

- Is the pipeline fully utilized? NO

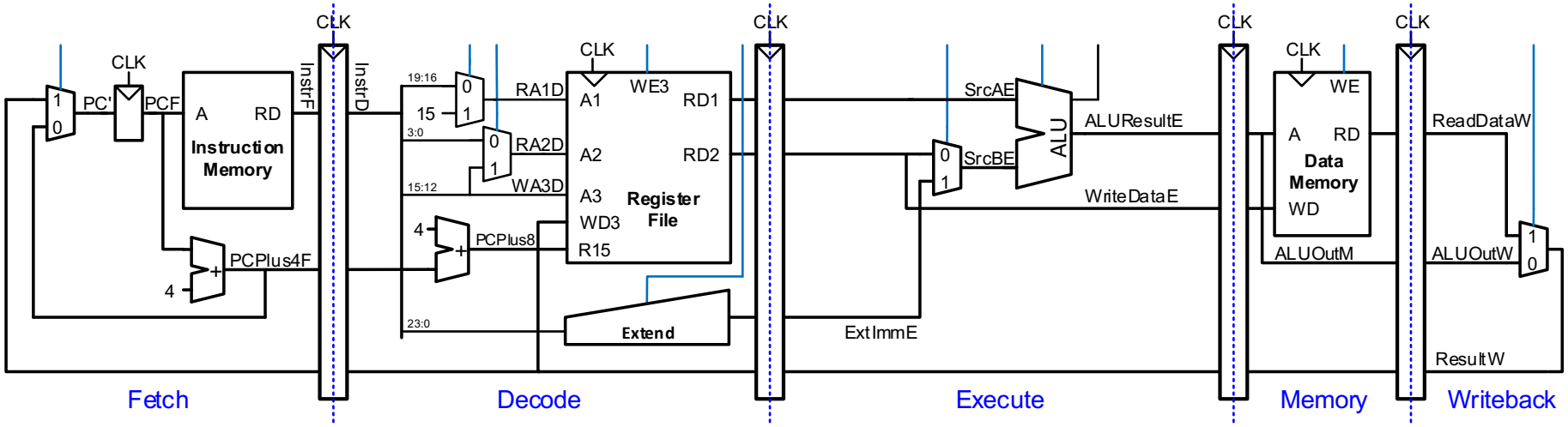
Pipeline Operation: Cycle 5



I5 I4 I3 I2 I1

- Is the pipeline fully utilized? YES

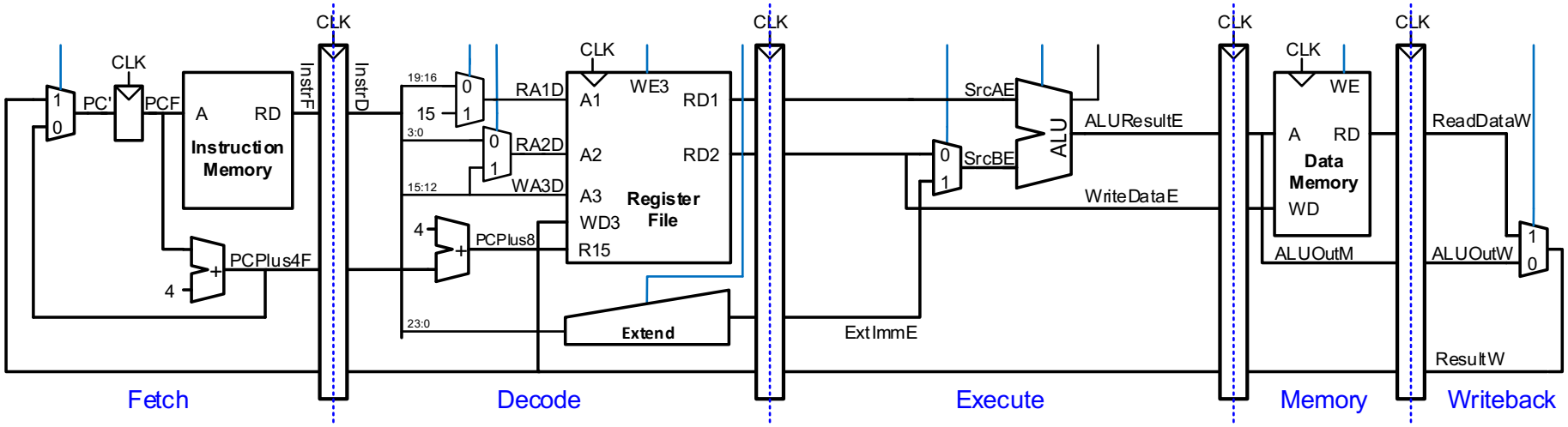
Pipeline Operation: Cycle 6



I6 I5 I4 I3 I2

- Is the pipeline fully utilized? YES

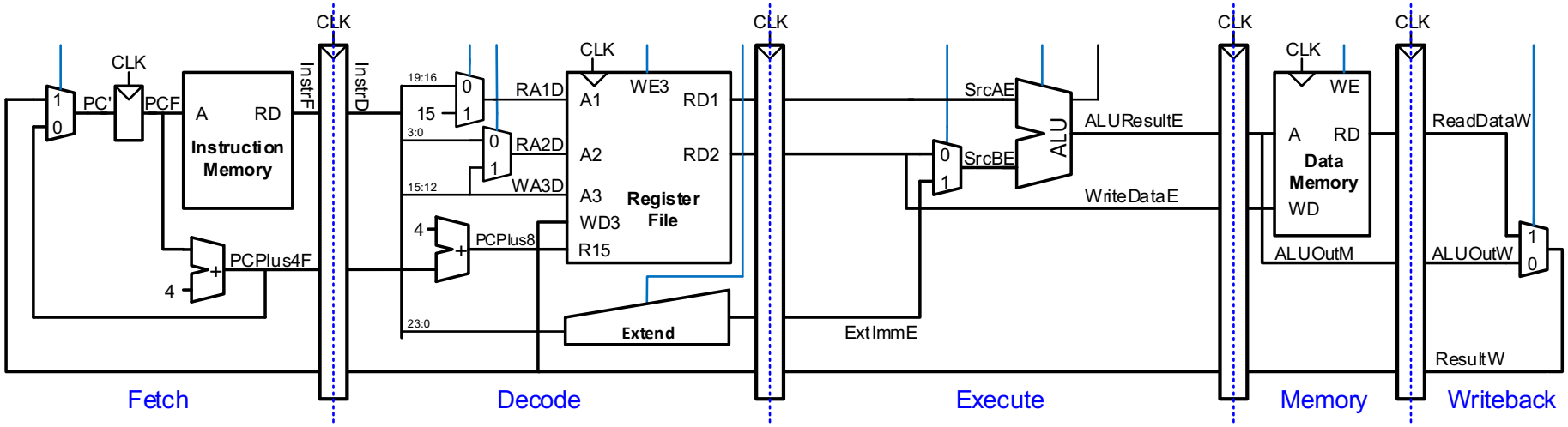
Pipeline Operation: Cycle 7



I6 I5 I4 I3

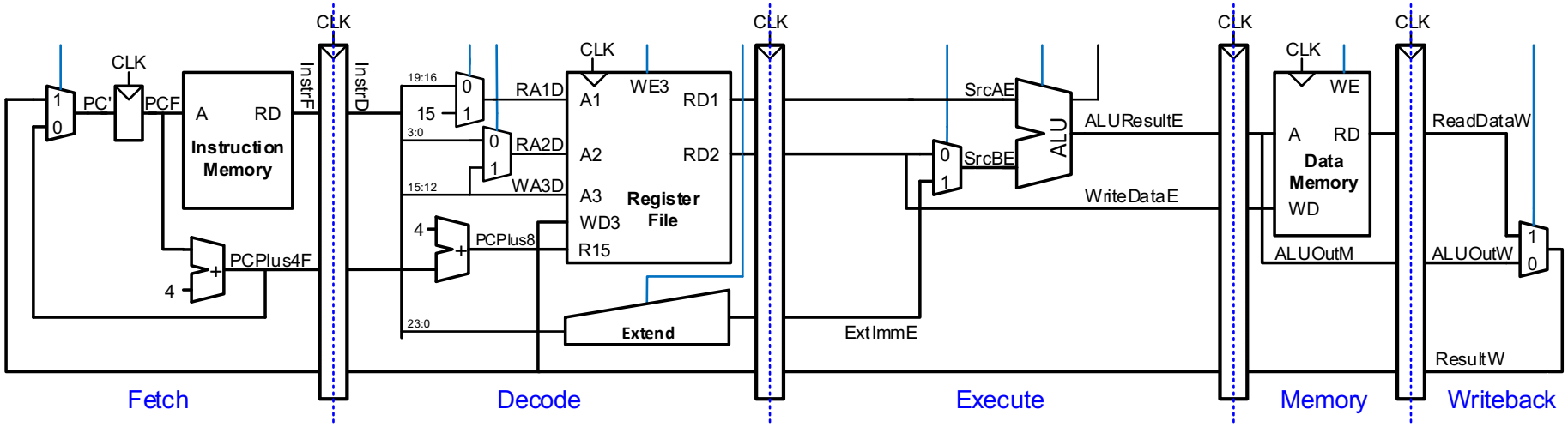
- Is the pipeline fully utilized? NO

Pipeline Operation: Cycle 8



- Is the pipeline fully utilized? NO

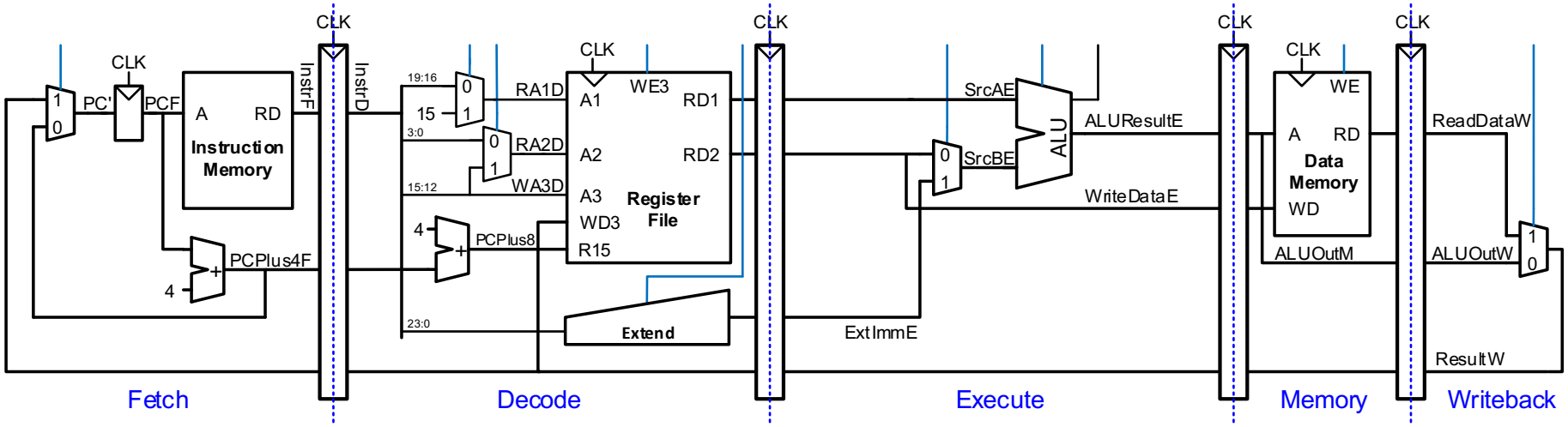
Pipeline Operation: Cycle 9



I6 I5

- Is the pipeline fully utilized? NO

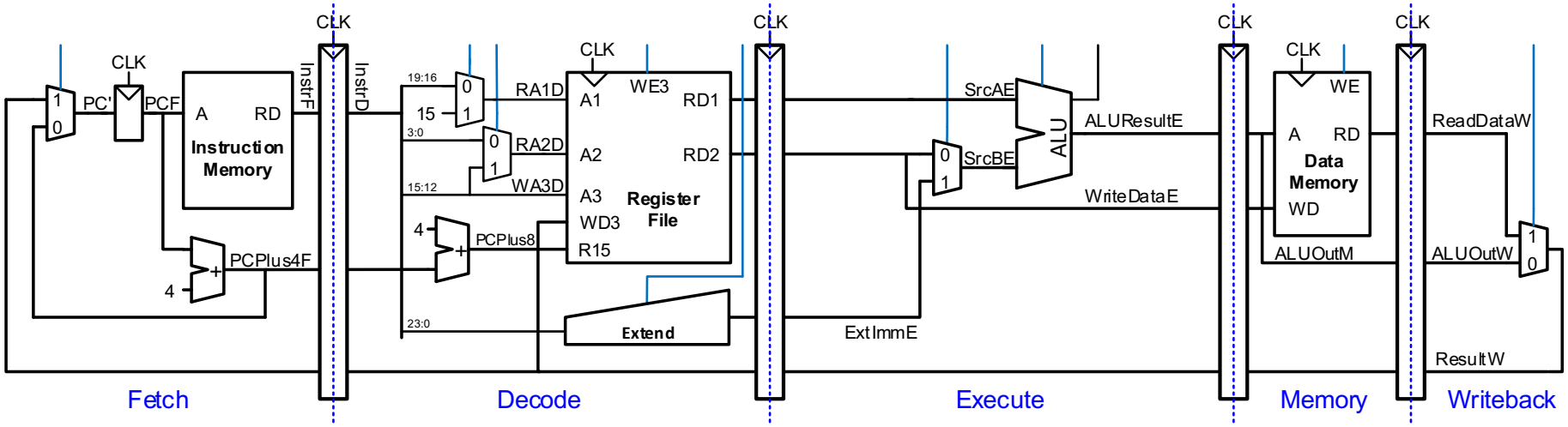
Pipeline Operation: Cycle 10



I6

- Is the pipeline fully utilized? NO

Pipeline Operation



- No more instructions to execute

Instruction-Level Parallelism

What is Instruction-Level Parallelism?

- Since 1985, all processors have used pipelining to overlap the execution of instructions to improve performance
  - This overlap is termed as instruction-level parallelism (ILP)

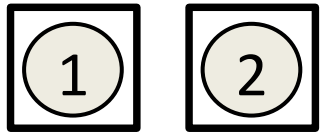
- The main limitation to exploiting high levels of ILP:
  - **Data and control dependences** in the program
  - Younger instructions "depend" on the results produced by older instructions

- **Historical (ongoing) debate: How best to exploit ILP?**
  - Dynamically in hardware (dynamic = during execution)
    - No need to recompile code, portable, transparent, hardware has more knowledge of program behavior: loop counters, inputs, branch behavior
    - Power, area, energy, security issues (end of Moore's law, transition to multicore)
  - Statically in software (find parallelism at compile time)
    - Compiler can do whole-program optimizations, inspired innovations in compiler technology, commercial failure ....

Example Sequence 1

1. load $r2, #0($r6)
2. add $r3, $r4, $r5

Two independent instructions



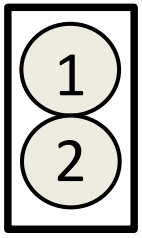
Example Sequence 2

1. load $r2, #0($r6)
2. add $r3, $r2, $r5

Data or true dependence

i2 needs the result of **i1**

A single (dependent) instruction chain



Example Sequence 3

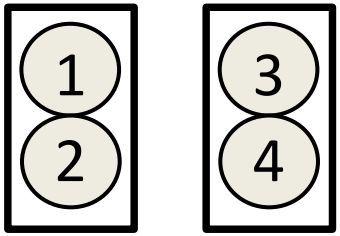
1. load $r2, #0($r6)
2. add $r3, $r2, $r5
3. load $r4, #0($r6)
4. add $r7, $r4, $r9

Data or true dependence

i2 needs the result of **i1**

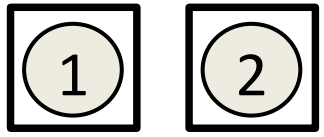
i4 needs the result of **i3**

Two independent instruction chains



Example Sequence 4

1. load $r2, #0($r6)
2. add $r6, $r3, $r5



False (anti) dependence

i2 needs to write what **i1** needs to read

Two independent instructions

Limitation of # registers, but ILP exists

*Rename register r6 in **i2** and execute in parallel*

Example Sequence 5

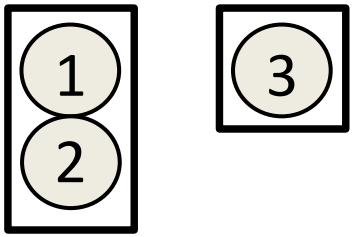
1. load $r2, #0($r6)
2. add $r3, $r2, $r5
3. add $r2, $r7, $r8

true dependence

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

*Rename register r2 in **i3** and execute in parallel with the chain*

Example Sequence 6

1. load $r2, #0($r6)
2. beq $r1, $r3, #BLAH
3. store $r2, #0($r9)
4. BLAH: store $r2, #0($r10)

Control dependence

Need to wait for the outcome of **i2** to fetch again

(Note: Ignoring any other dependences)

Limitation of control-flow architecture: branches
If we can guess the branch outcome, we can fetch from the correct path without waiting for the branch to execute

Pipeline Hazards

- When multiple instructions are handled concurrently there is a danger of hazard

- Hazards are a part of real life

- Some **coping strategies:** Get around, precaution, mitigate harm after

Pipeline Hazards (Three Types)

- Structural hazard
  - When two instructions want to use the same resource
  - Memory for instructions (F) and data (M)
  - Register file is accessed in two different stages (**what are those?**)
- Data hazard
  - When a dependent instruction wants the result of an earlier instruction
- Control hazard
  - When a PC-changing instruction is in the pipeline (**why is this a hazard?**)

Dependences and Hazards

- Dependence is a program's property
- Hazard is a microarchitecture property

- True dependence results in:
  - Read-after-write hazard (RAW)

- (Name) Anti-dependence results in:
  - Write-after-read hazard (WAR)

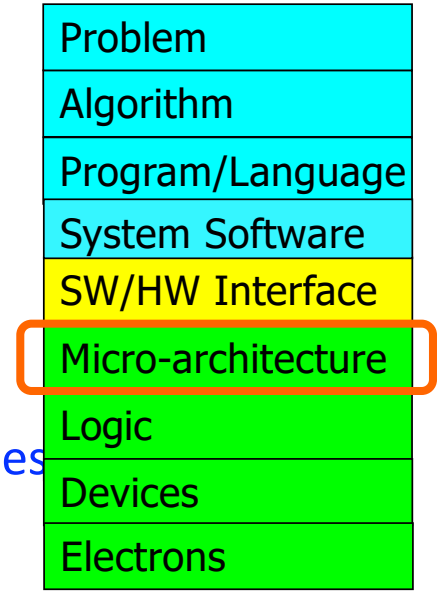
- (Name) Output dependence results in:
  - Write-after-write hazard (WAW)

- **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!

Exploiting ILP: A Taxonomy

Different Approaches, One Goal

- Ultimate aim of ILP machine
  - Issue multiple instructions in a clock cycle
- How can we do it?
  - Computer = hardware + software
  - Who can discover ILP more efficiently?
    - **Qualified question:** Who can find interesting instruction schedules efficiently?
- Three competing approaches
  - **Statically scheduled superscalar processors**
  - **VLIW (very long instruction word processors)**
  - **Dynamically scheduled superscalar processors**
- VLIW had success in embedded domain. Dynamic scheduling went BIG everywhere!



Instruction Scheduling

- **Statically scheduled superscalar processor**
  - Compiler schedules instructions during program creation
  - Hardware does no reordering of instructions (sequential unless branch changes PC)
  - Compiler can create "interesting schedules" by doing deep program analysis
  - Schedule is **static** as it does not change dynamically based on different outcomes of a branch

- **VLIW (Very Long Instruction Word) processor**
  - Static scheduling by compiler. Instruction words are very large. Up to 28 insts. in a bundle
  - Compiler does "smart" analysis to construct "interesting" schedules (interesting = high ILP)
  - Conceptually the same as above. "Some differences" in philosophy (smart compiler, dumb hw.)

- **Dynamically scheduled superscalar processor**
  - Hardware does scheduling during program execution
  - Can reorder instructions to extract maximum ILP
  - Hardware can construct different "instruction schedules" based on different executions of the same set of basic blocks (different branch outcomes)

VLIW Philosophy & Principles

Proceedings of the ACM SIGPLAN '84 Symposium on Compiler Construction,
SIGPLAN Notices Vol. 19, No. 6, June 1984

Parallel Processing: A Smart Compiler and a Dumb Machine

**Joseph A. Fisher, John R. Ellis,
John C. Ruttenberg, and Alexandru Nicolau**

**Department of Computer Science, Yale University
New Haven, CT 06520**

Abstract

Multiprocessors and vector machines, the only successful parallel architectures, have coarse-grained parallelism that is hard for compilers to take advantage of. We've developed a new fine-grained parallel architecture and a compiler that together offer order-of-magnitude speedups for ordinary scientific code.

future, and we're building a VLIW machine, the ELI (Enormously Long Instructions) to prove it.

In this paper we'll describe some of the compilation techniques used by the Bulldog compiler. The ELI project and the details of Bulldog are described elsewhere [4, 6, 7, 15, 17].

Fisher et al., "Parallel Processing: A Smart Compiler and a Dumb Machine," CC 1984.

Commercial VLIW Machines

- Multiflow TRACE, Josh Fisher (7-wide, 28-wide)
- Cydrome Cydra 5, Bob Rau
- Transmeta Crusoe: x86 binary-translated into internal VLIW
- TI C6000, Trimedia, STMicro (DSP & embedded processors) and some ATI/AMD GPUs
  - Most successful commercially

- Intel IA-64
  - Not fully VLIW, but based on VLIW principles
  - EPIC (Explicitly Parallel Instruction Computing)
  - Instruction bundles can have dependent instructions
  - A few bits in the instruction format specify explicitly which instructions in the bundle are dependent on which other ones

In-Order Pipeline: The Problem

Baseline In-Order Pipeline

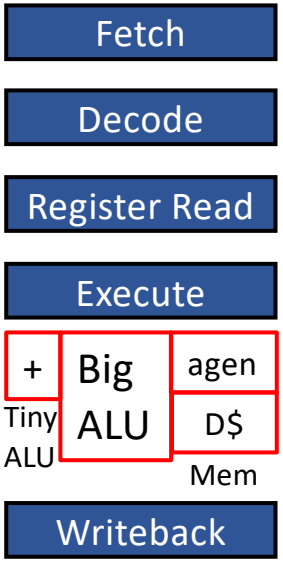
- Let’s first establish that the problem with in-order pipeline is not resource limitation
- The problem is in the issue policy
- Let’s take an aggressive in-order pipeline
  - Non-blocking execute stage
  - Have as many functional units as required

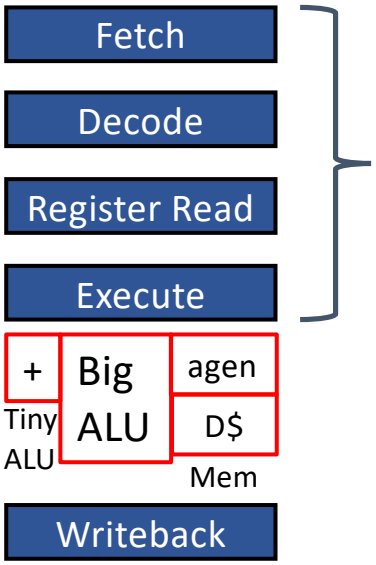
Baseline In-Order Pipeline

- **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

Baseline In-Order Pipeline

- **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)
  - Register renaming (optional optimization)

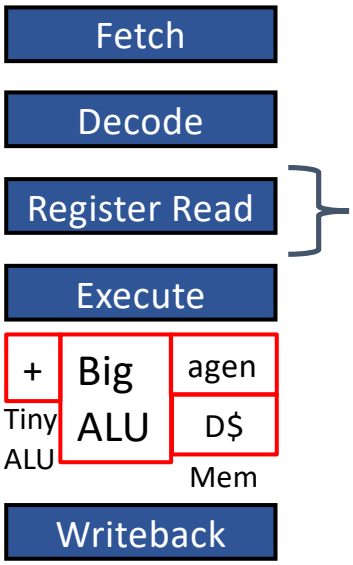

Fetch
Decode
Register Read
Execute
+
Big
ALU
agen
D$
Tiny
ALU
Mem
Writeback



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

Scenario 1: load miss followed by independent instructions

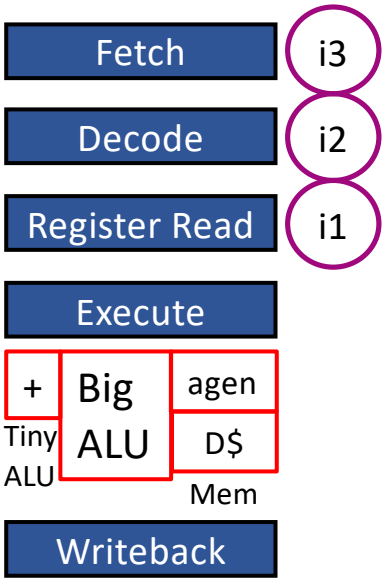
```
i1: load   r2,   #0(r1)
i2: add    r4,   r3, #1
i3: add    r6,   r5, #2
i4: add    r7,   r6, #3
```

	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: load   r2,   #0(r1)
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: load   r2,   #0(r1)
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: load   r2,   #0(r1)
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: load   r2,   #0(r1)
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

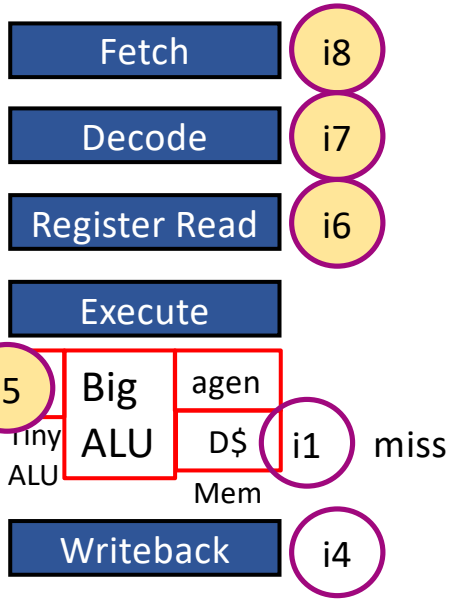
```
i1: load   r2,   #0(r1)
i2: add    r4,   r3, #1
i3: add    r6,   r5, #2
i4: add    r7,   r6, #3
```

	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: load   r2,   #0(r1)
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: load   r2,   #0(r1)
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: load   r2,   #0(r1)
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 2: Load miss followed by dependent instruction, followed by independent instructions

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

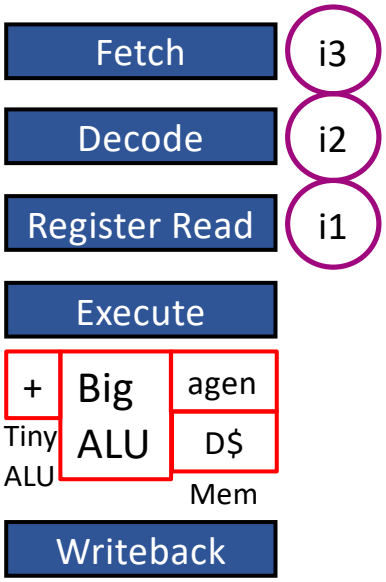
```
i1: load  r2,  #0(r1)
i2: add   r4,  r2, #1
i3: add   r6,  r5, #2
i4: add   r7,  r6, #3
```

	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: load  r2,  #0(r1)
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

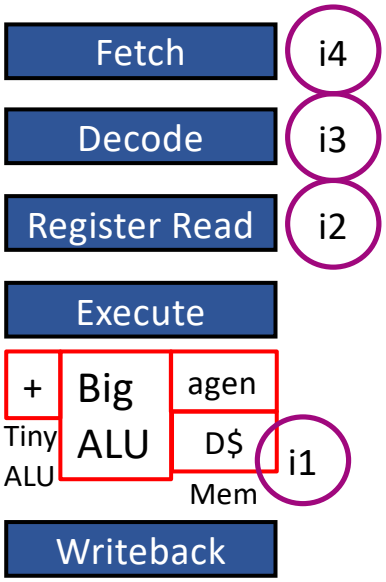
```
i1: load  r2,  #0(r1)
i2: add   r4,  r2, #1
i3: add   r6,  r5, #2
i4: add   r7,  r6, #3
```

	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: load   r2,   #0(r1)
i2: add    r4,   r2, #1
i3: add    r6,   r5, #2
i4: add    r7,   r6, #3
```

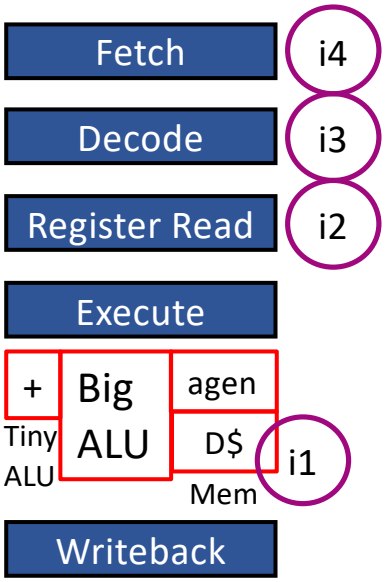
	1	2	3	4	5	6	7	8	9	10	11	12	13
i1	FE	DE	RR	$EX_{@}$									
i2		FE	DE	RR									
i3			FE	DE									
i4				FE									



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

```
i1: load  r2,  #0(r1)
i2: add   r4,  r2, #1
i3: add   r6,  r5, #2
i4: add   r7,  r6, #3
```

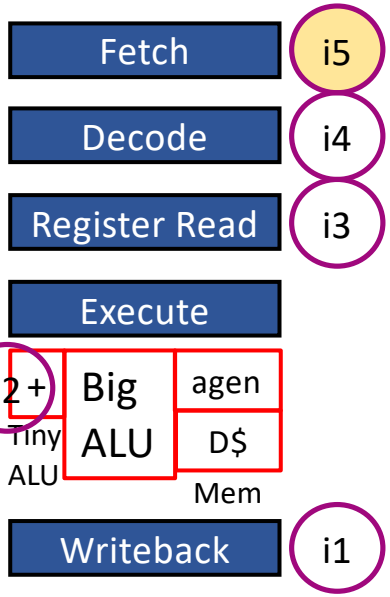
	1	2	3	4	5	6	7	8	9	10	11	12	13
i1	FE	DE	RR	$EX_{@}$	$EX_{D\$}$	...miss...							
i2		FE	DE	RR	RR								
i3			FE	DE	DE								
i4				FE	FE								



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

```
i1: load  r2,  #0(r1)
i2: add   r4,  r2, #1
i3: add   r6,  r5, #2
i4: add   r7,  r6, #3
```

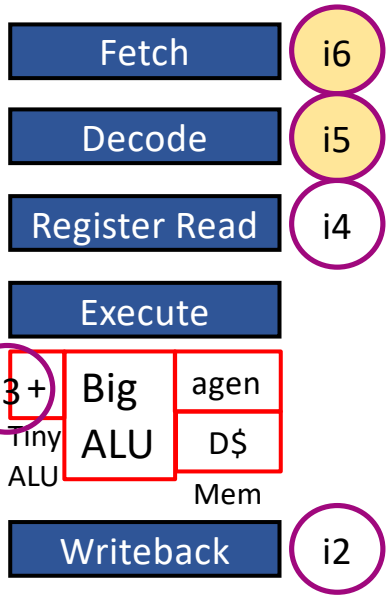
	1	2	3	4	5	6	7	8	9	10	11	12	13
i1	FE	DE	RR	$EX_{@}$	$EX_{D\$}$	...miss...							
i2		FE	DE	RR	RR	RR	RR	RR	RR				
i3			FE	DE	DE	DE	DE	DE	DE				
i4				FE	FE	FE	FE	FE	FE				



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

```
i1: load  r2,  #0(r1)
i2: add   r4,  r2, #1
i3: add   r6,  r5, #2
i4: add   r7,  r6, #3
```

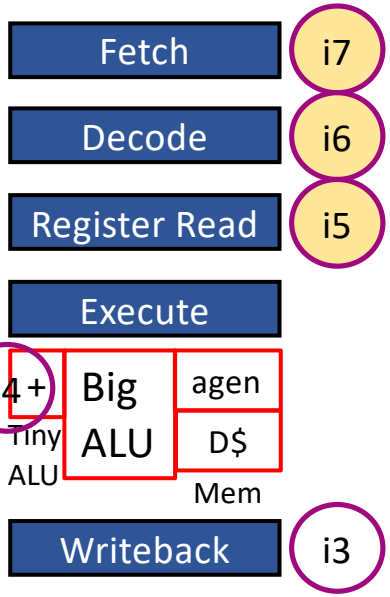
	1	2	3	4	5	6	7	8	9	10	11	12	13
i1	FE	DE	RR	$EX_{@}$	$EX_{D\$}$	...miss...				WB			
i2		FE	DE	RR	RR	RR	RR	RR	RR	EX			
i3			FE	DE	DE	DE	DE	DE	DE	RR			
i4				FE	FE	FE	FE	FE	FE	DE			



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

```
i1: load  r2,  #0(r1)
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: load  r2,  #0(r1)
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: load  r2,  #0(r1)
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

In-Order Issue Bottleneck

- i2 must wait for i1
  - i2 depends on i1 (chain of dependent instructions)
- i3, i4 need not wait for the i1-i2 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*

OOO pipeline unblocks RR (issue) using a new instruction buffer for stalled data-dependent instructions

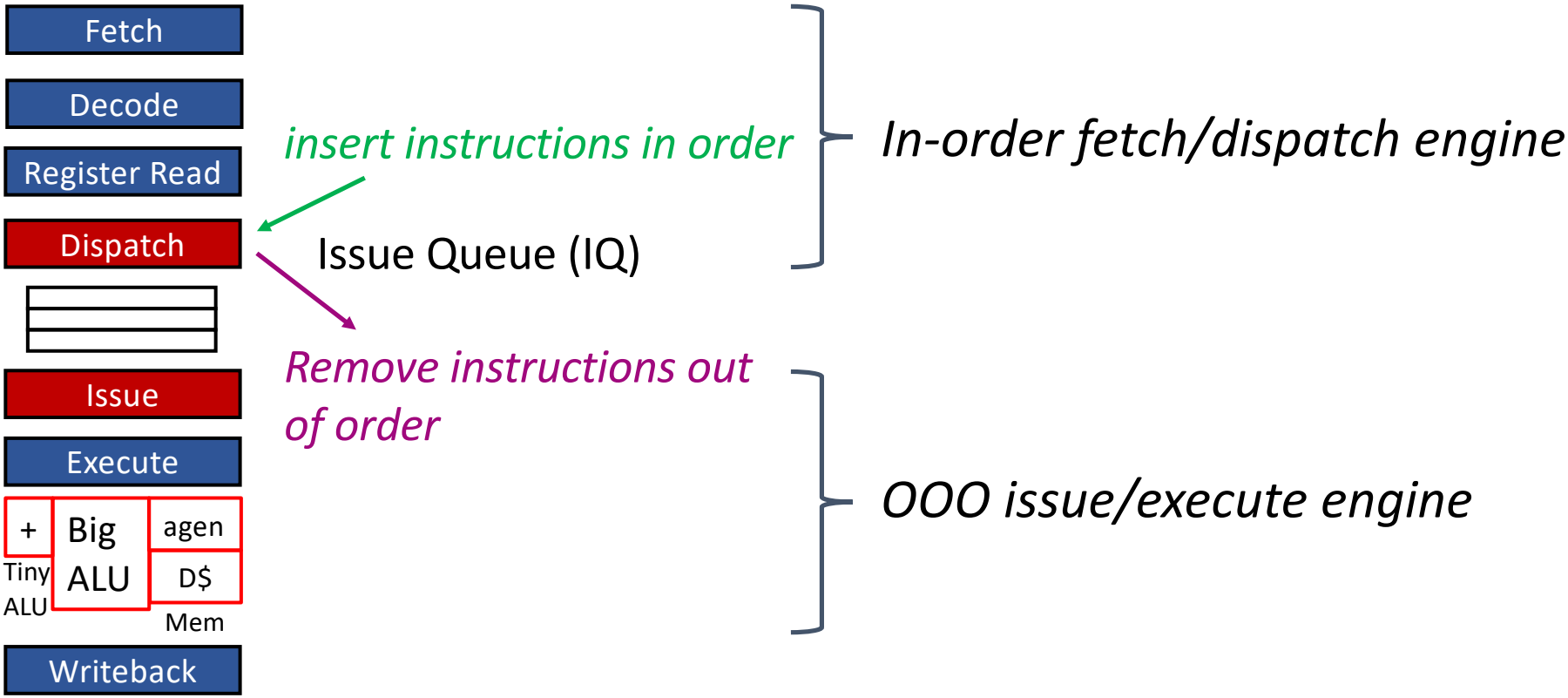
- *A structure with many names: "Reservation stations", "issue buffer", "issue queue", "scheduler", "scheduling window"*

From In-Order to Out-of-Order

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)



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

Two Humps in a Modern Pipeline

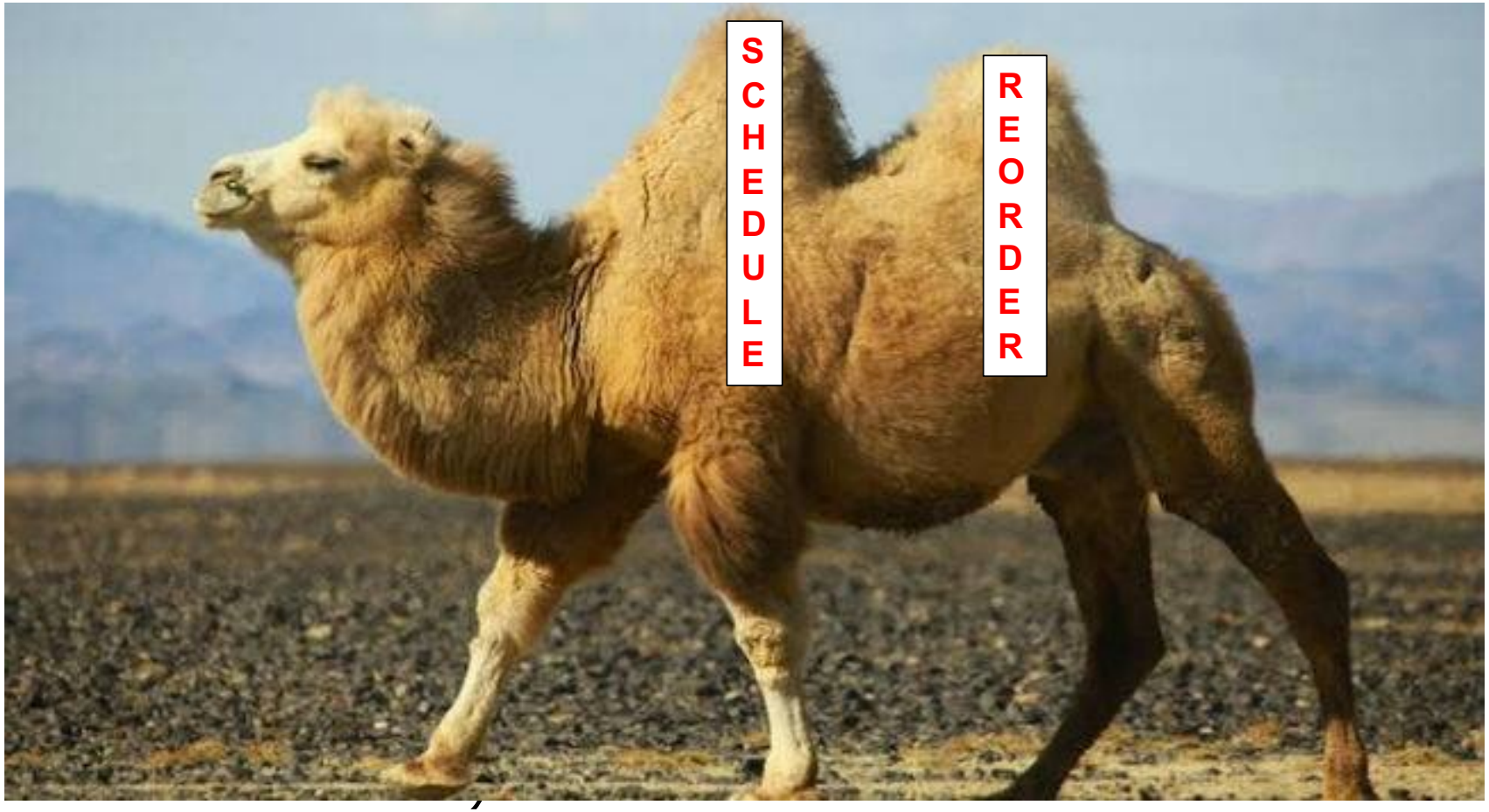
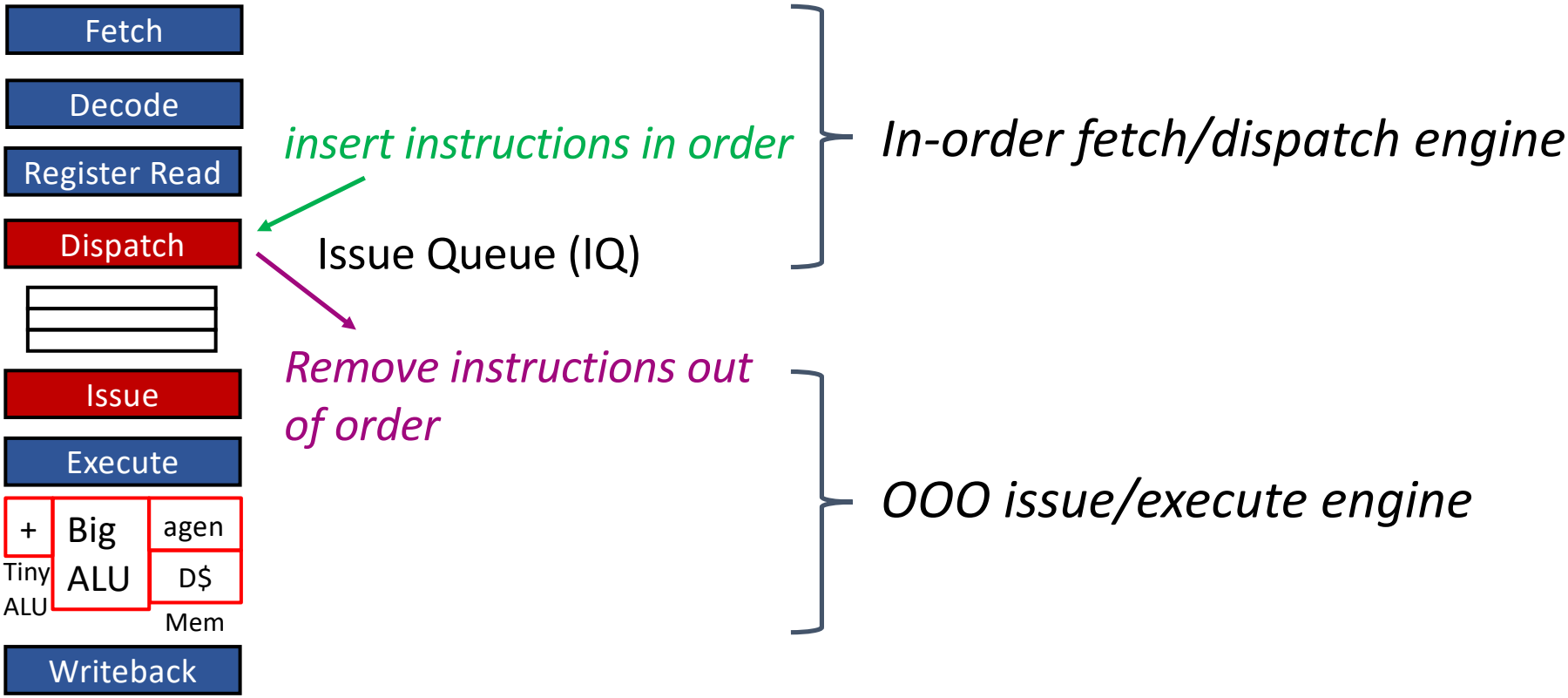
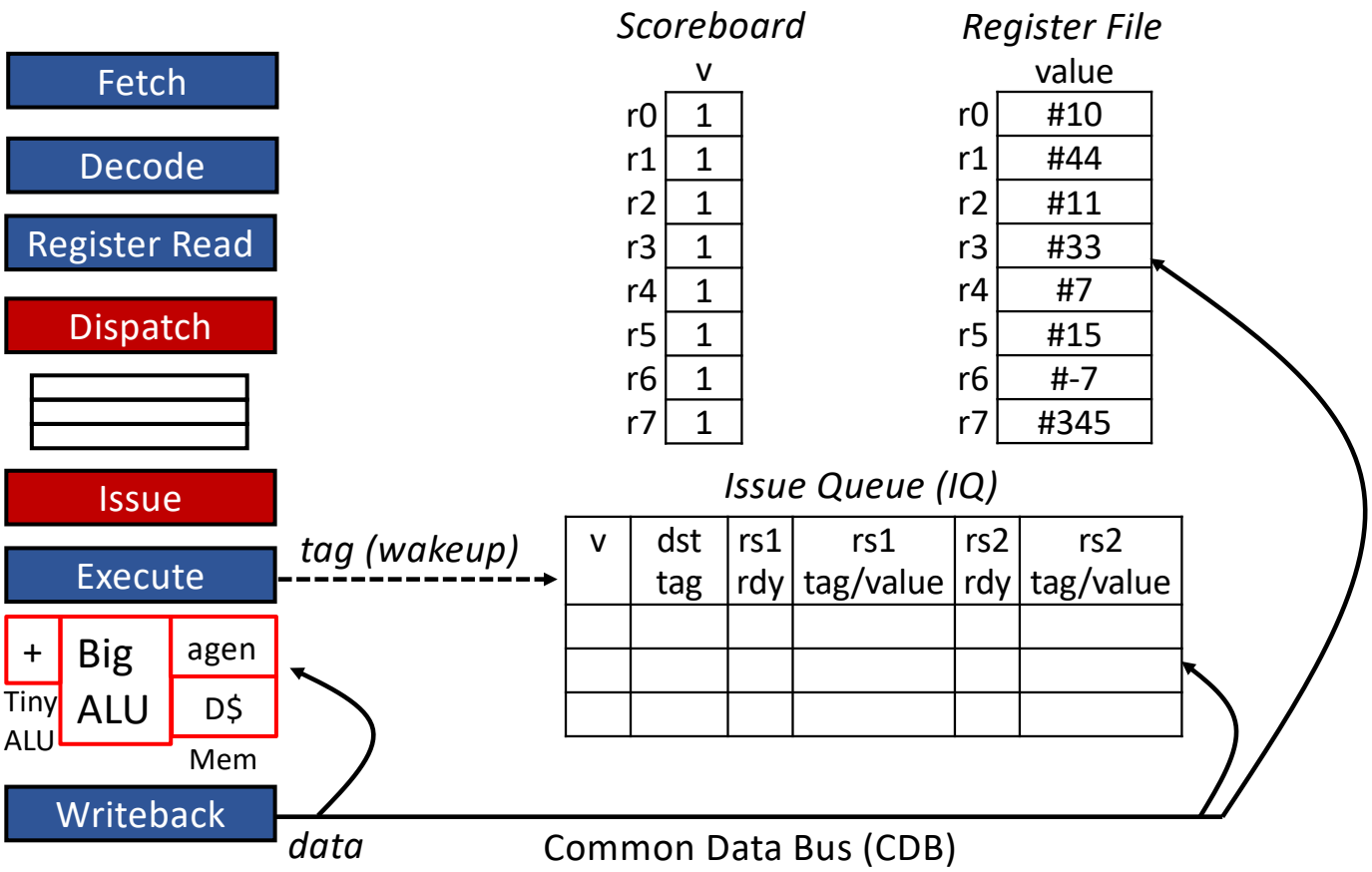


Photo credit: http://true-wildlife.blogspot.ch/2010/10/bactrian-camel.html

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





Dispatch 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)

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

CDC 6600 style scoreboard:

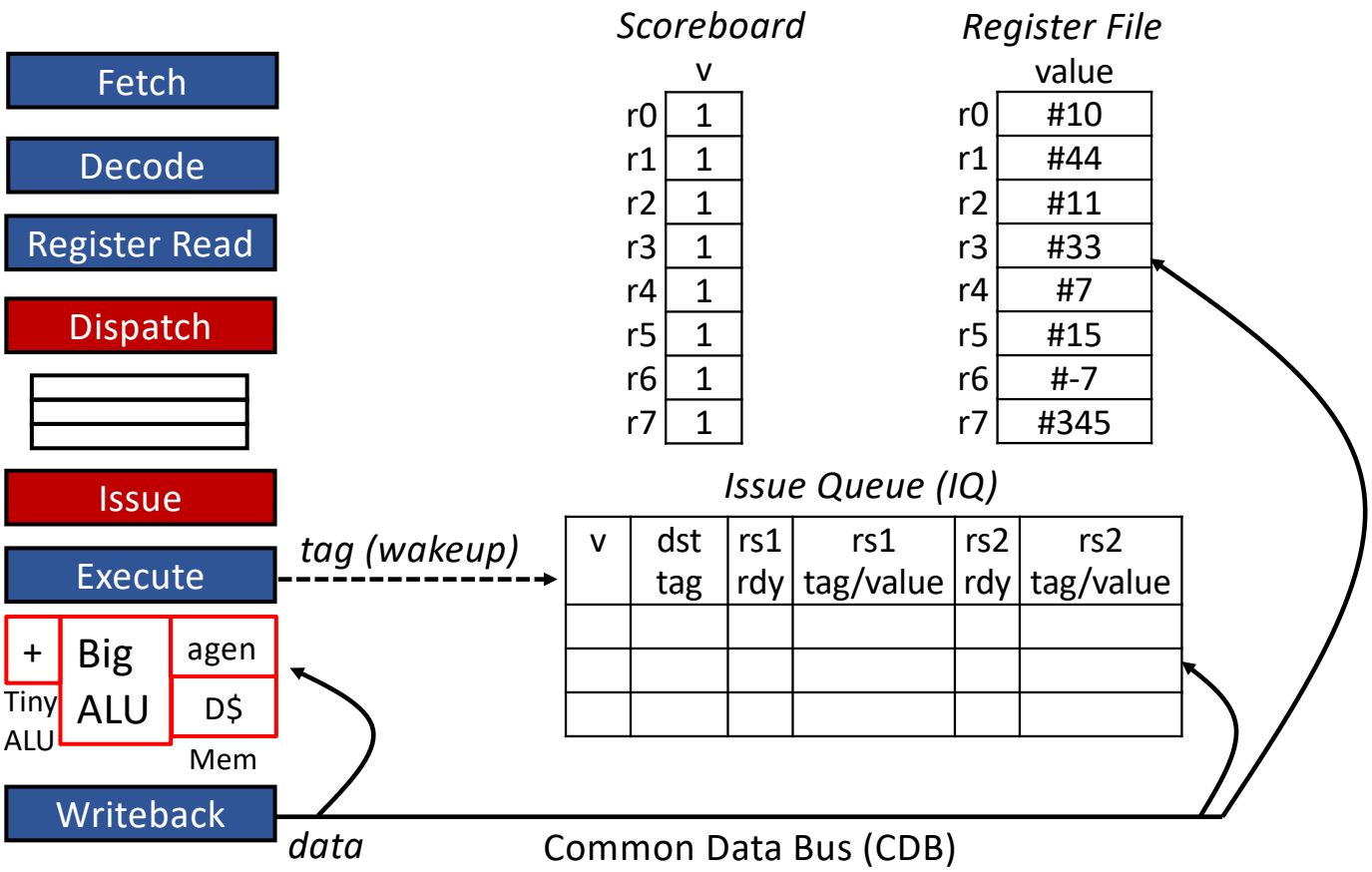
- When an instruction has register rN as a destination (N=0-7), set the corresponding bit to 0 (busy)
- Instructions capture the tag if v=1 (busy) and value otherwise (from RF)

Instruction wakeup and select:

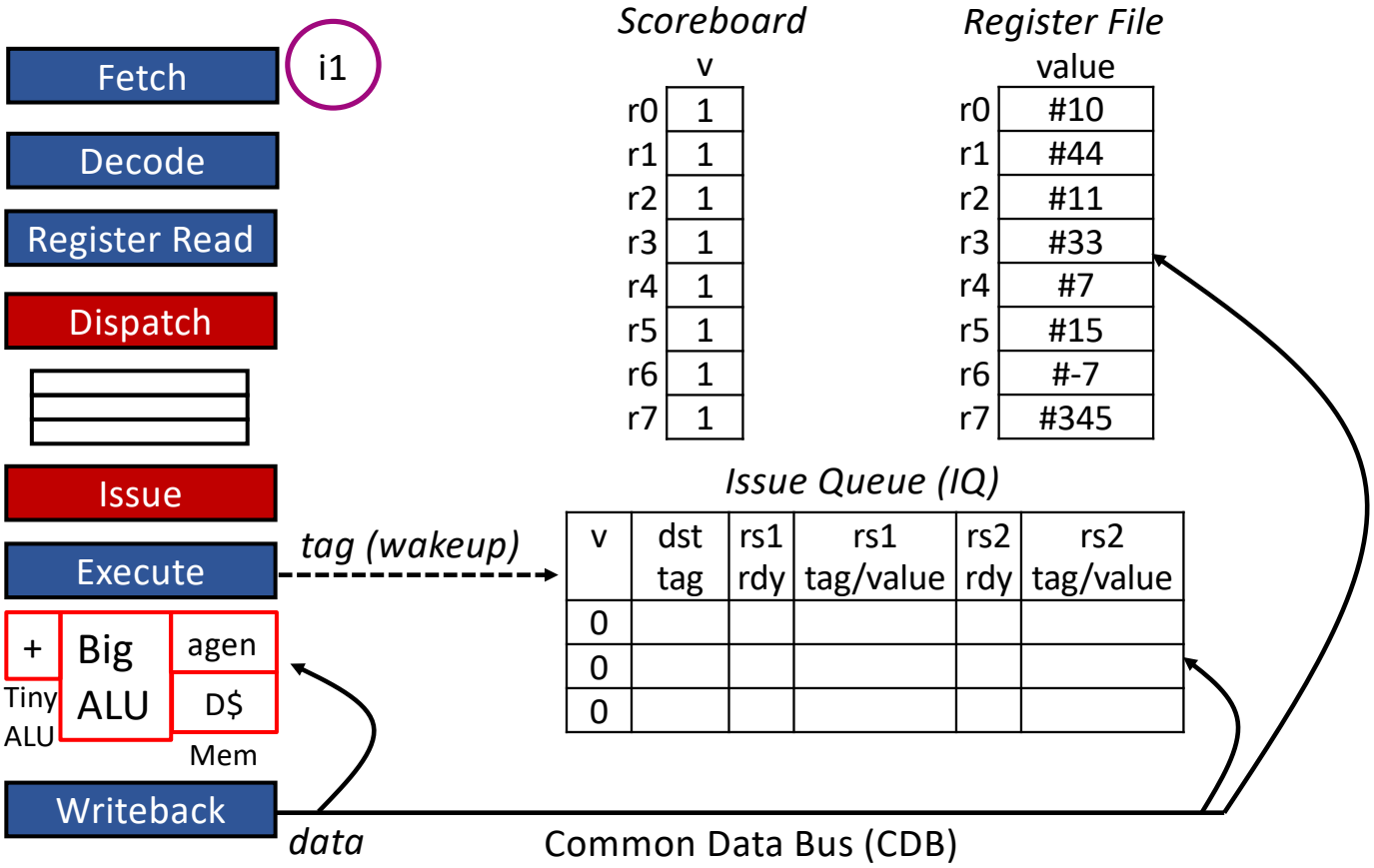
- The *wakeup logic* in front of the issue queue *snoops* for destination tags of parent instructions. When the destination tag appears, it wakes up all instructions waiting for that tag.
- X: tag, X+1: value, capture-tag-and-go
- The *selection logic* in the issue stage decides which of the "ready" instructions to execute next.

Forwarding via the CDB

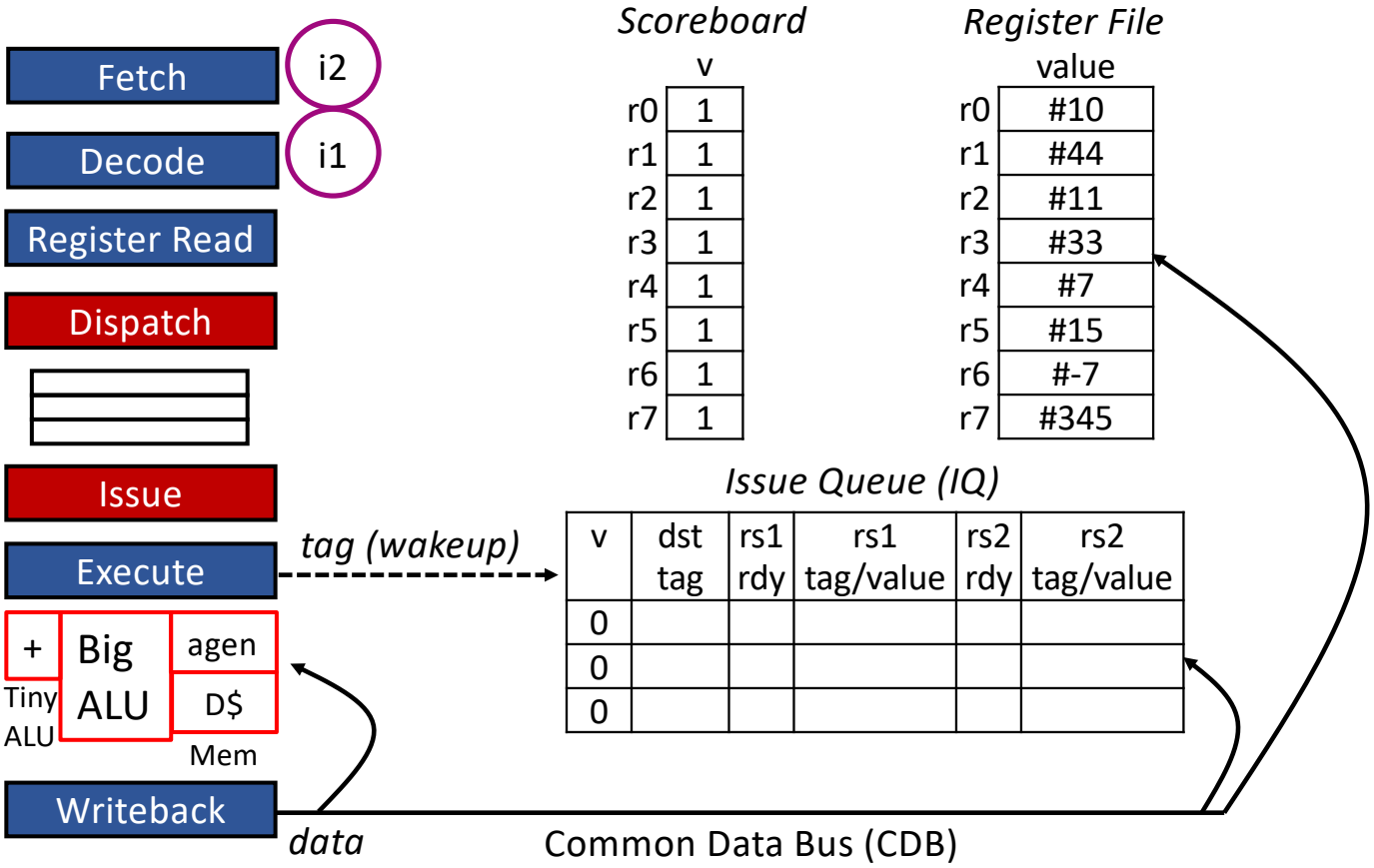
- The values are broadcasted over the common data bus bypassing register file writes. This bus resembles the forwarding/bypass network in the MIPS pipeline



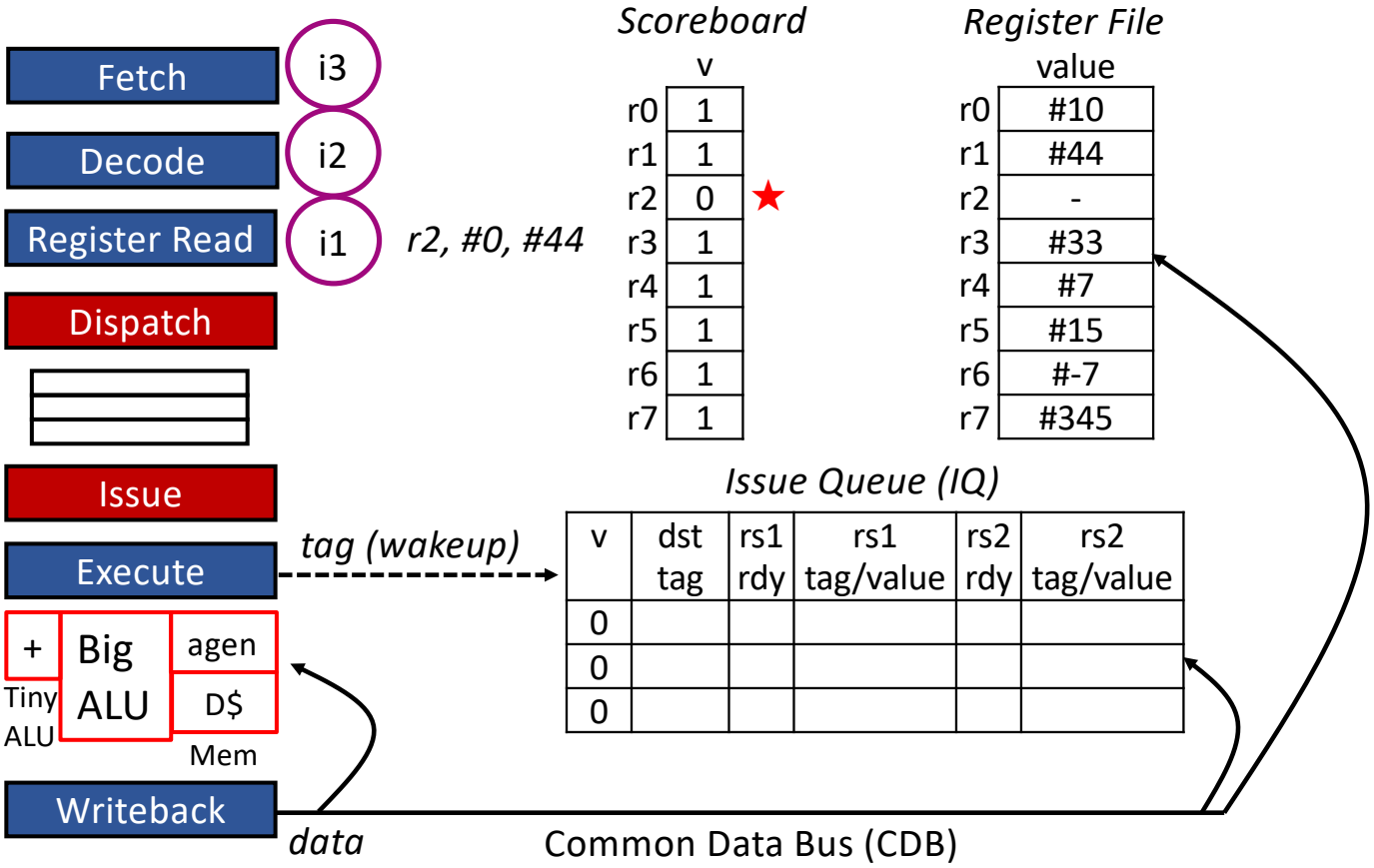
	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
i1: load r2, #0(r1)																	
i2: add r4, r2, #1																	
i3: add r6, r5, #2																	
i4: add r7, r6, #3																	



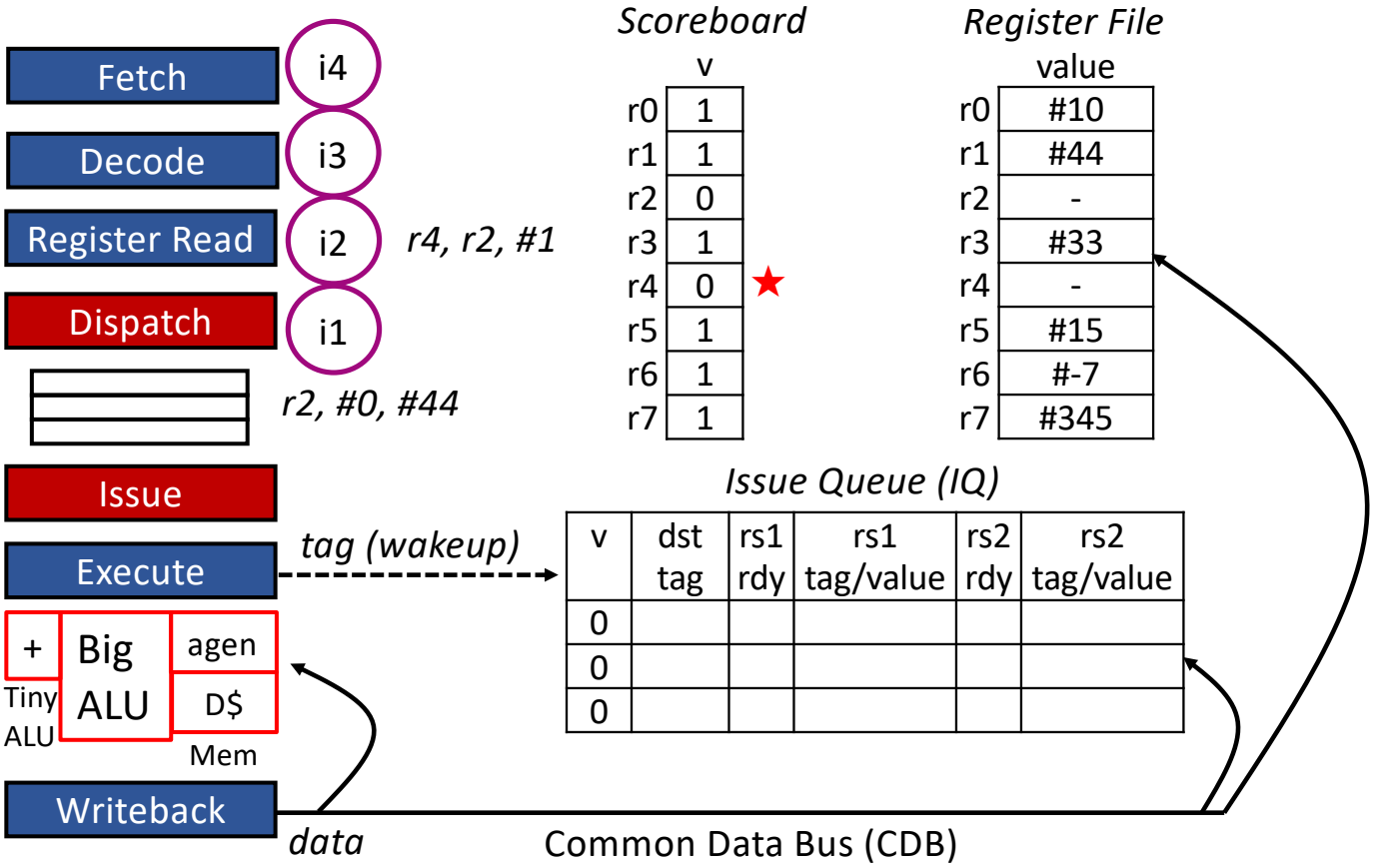
	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
i1: load r2, #0(r1)	FE																
i2: add r4, r2, #1																	
i3: add r6, r5, #2																	
i4: add r7, r6, #3																	



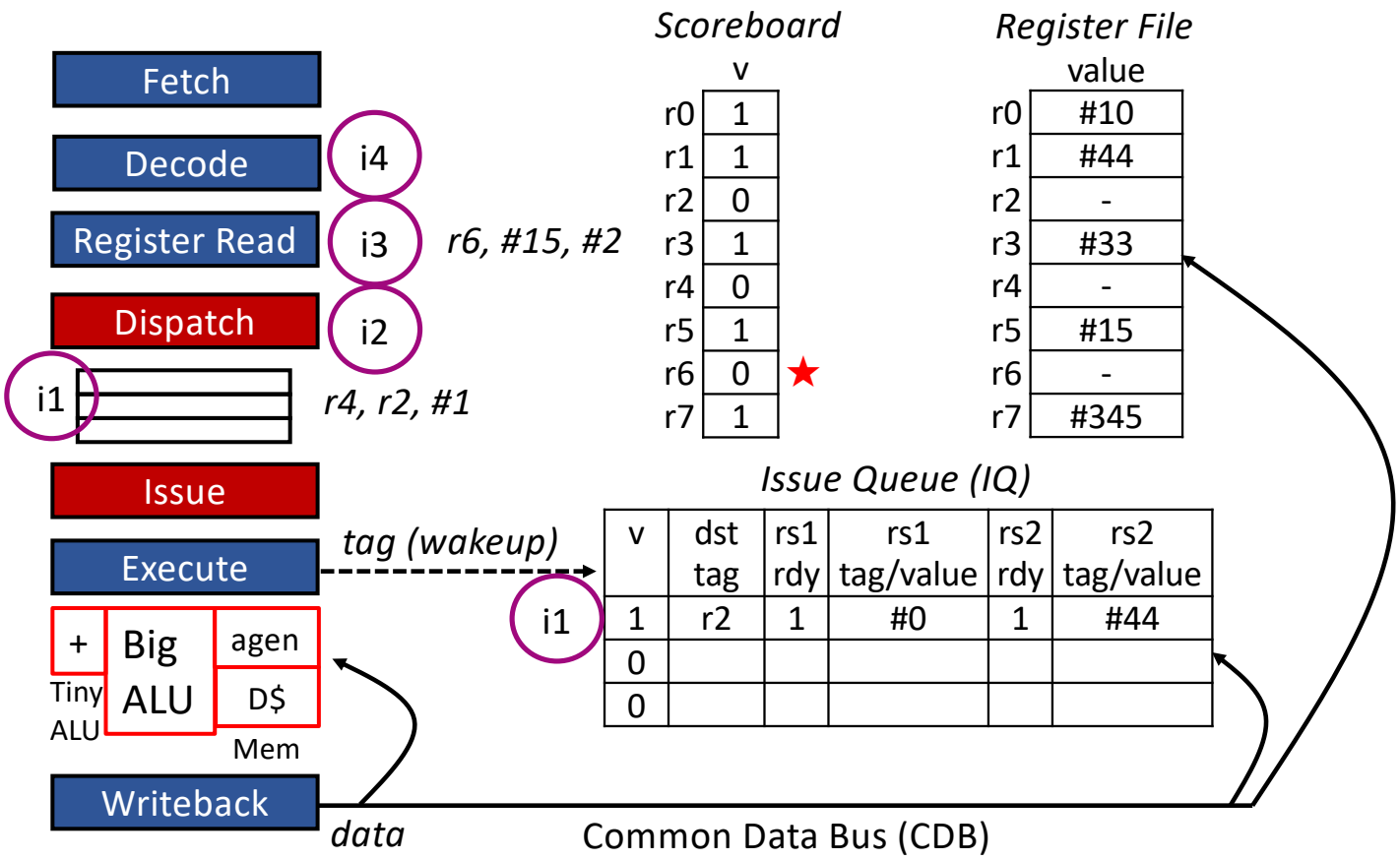
	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
i1: load r2, #0(r1)	FE	DE															
i2: add r4, r2, #1		FE															
i3: add r6, r5, #2																	
i4: add r7, r6, #3																	



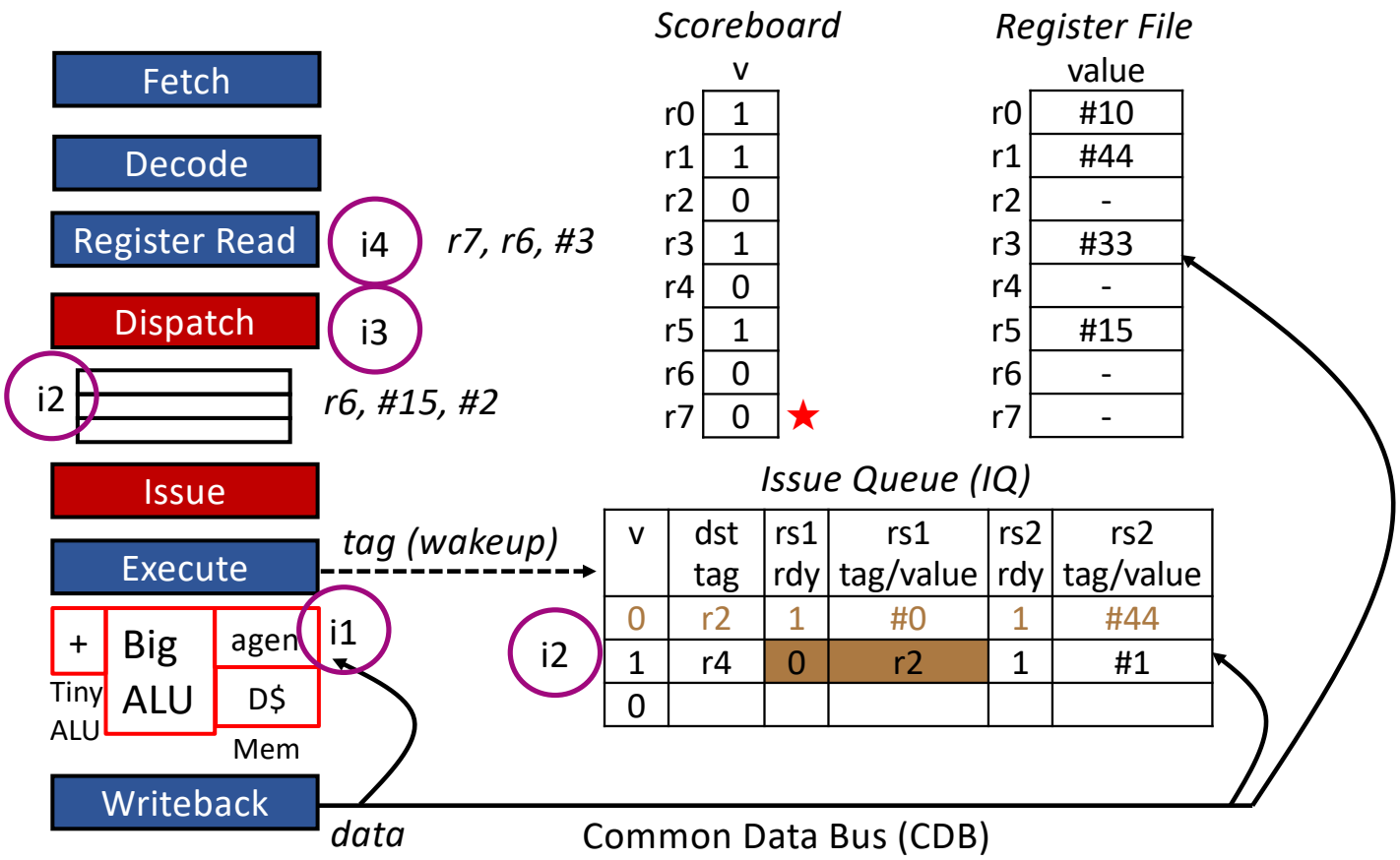
	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
i1: load r2, #0(r1)	FE	DE	RR														
i2: add r4, r2, #1		FE	DE														
i3: add r6, r5, #2			FE														
i4: add r7, r6, #3																	



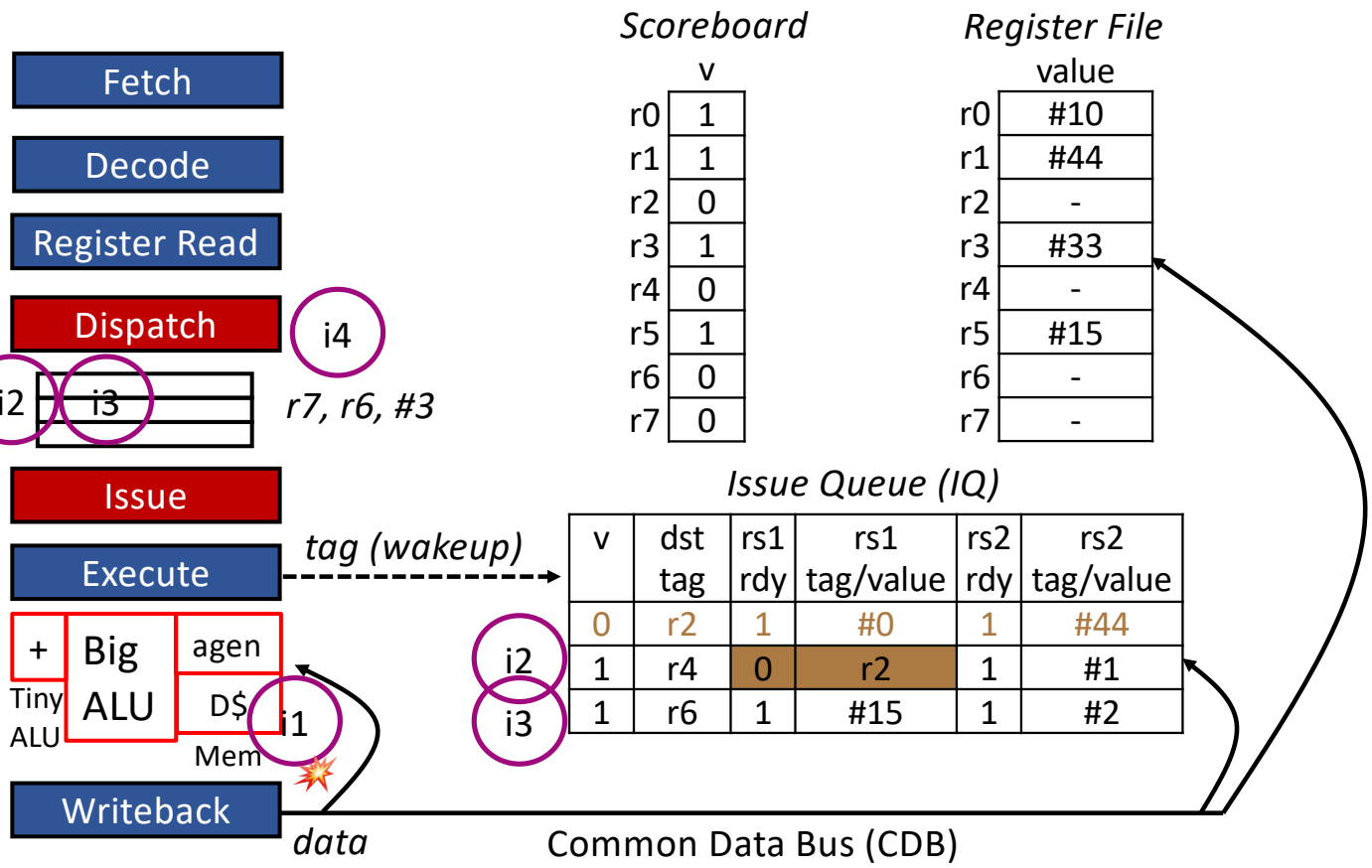
	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
i1: load r2, #0(r1)	FE	DE	RR	DI													
i2: add r4, r2, #1		FE	DE	RR													
i3: add r6, r5, #2			FE	DE													
i4: add r7, r6, #3				FE													



	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
i1: load r2, #0(r1)	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												

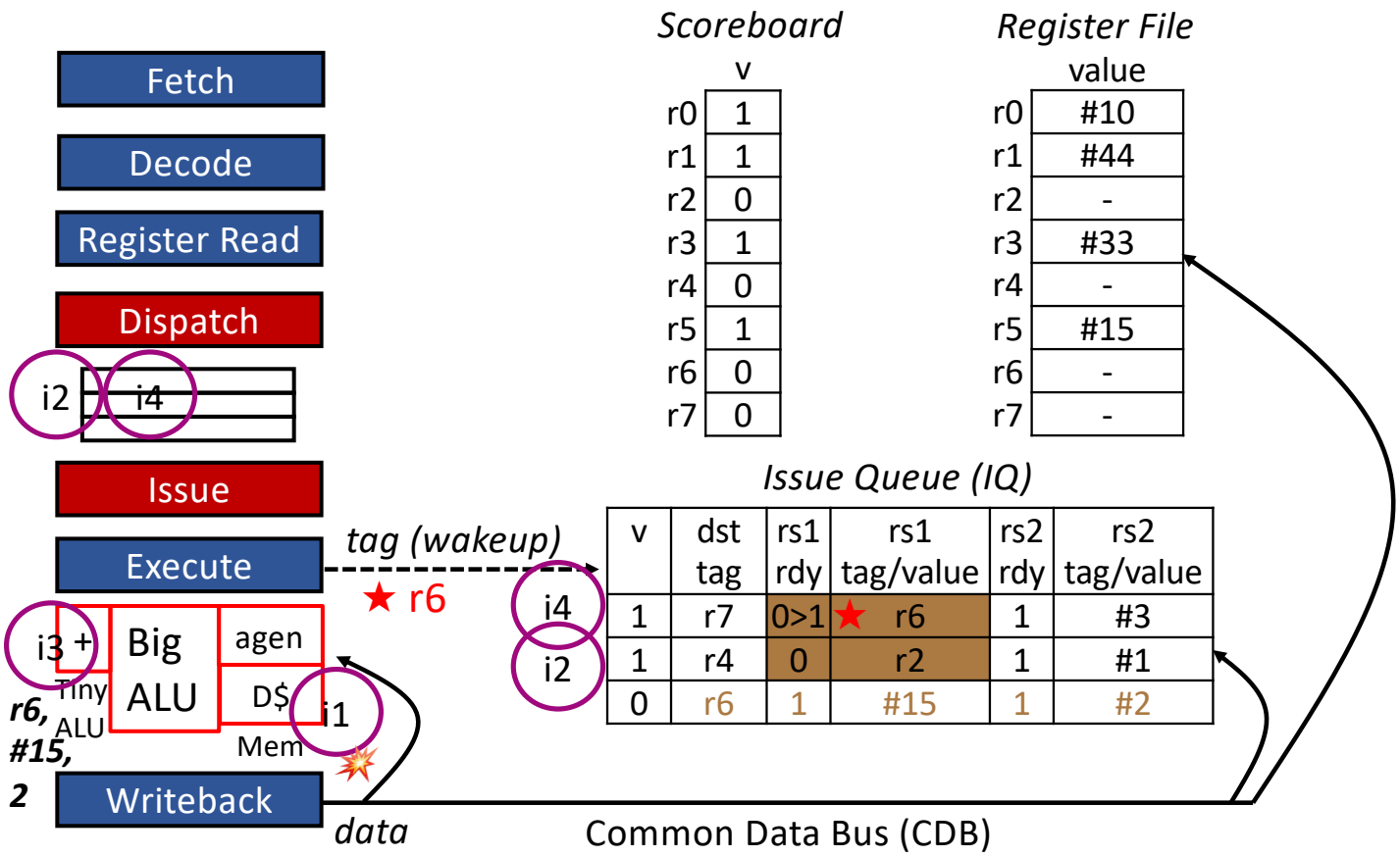


	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
i1: load r2, #0(r1)	FE	DE	RR	DI	IS	$EX_{@}$											
i2: add r4, r2, #1		FE	DE	RR	DI	IS											
i3: add r6, r5, #2			FE	DE	RR	DI											
i4: add r7, r6, #3				FE	DE	RR											



cache miss
r2, @44

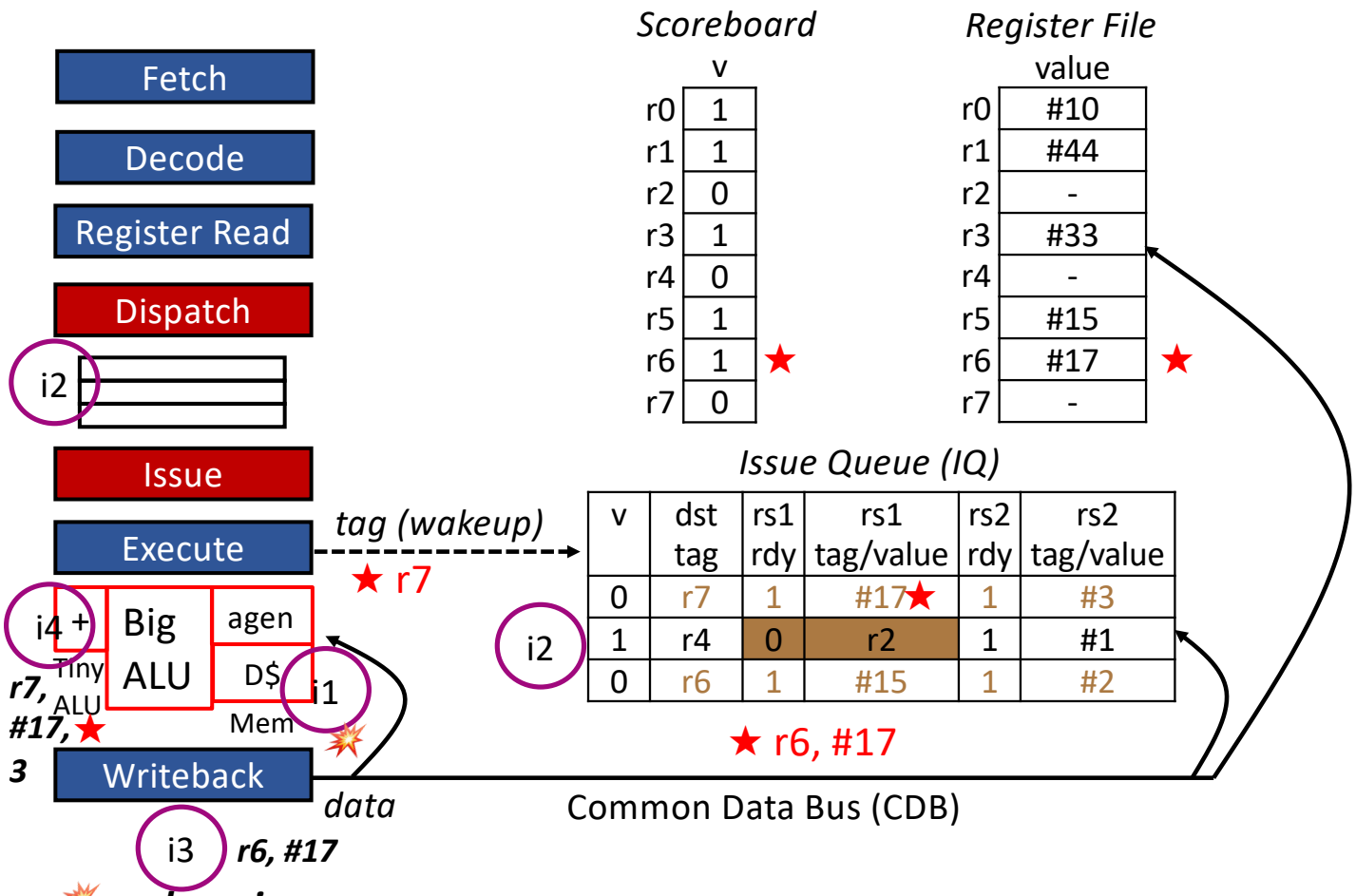
	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
i1: load r2, #0(r1)	FE	DE	RR	DI	IS	$EX_{@}$	$EX_{D\$}$	... miss ...									
i2: add r4, r2, #1		FE	DE	RR	DI	IS	IS										
i3: add r6, r5, #2			FE	DE	RR	DI	IS										
i4: add r7, r6, #3				FE	DE	RR	DI										



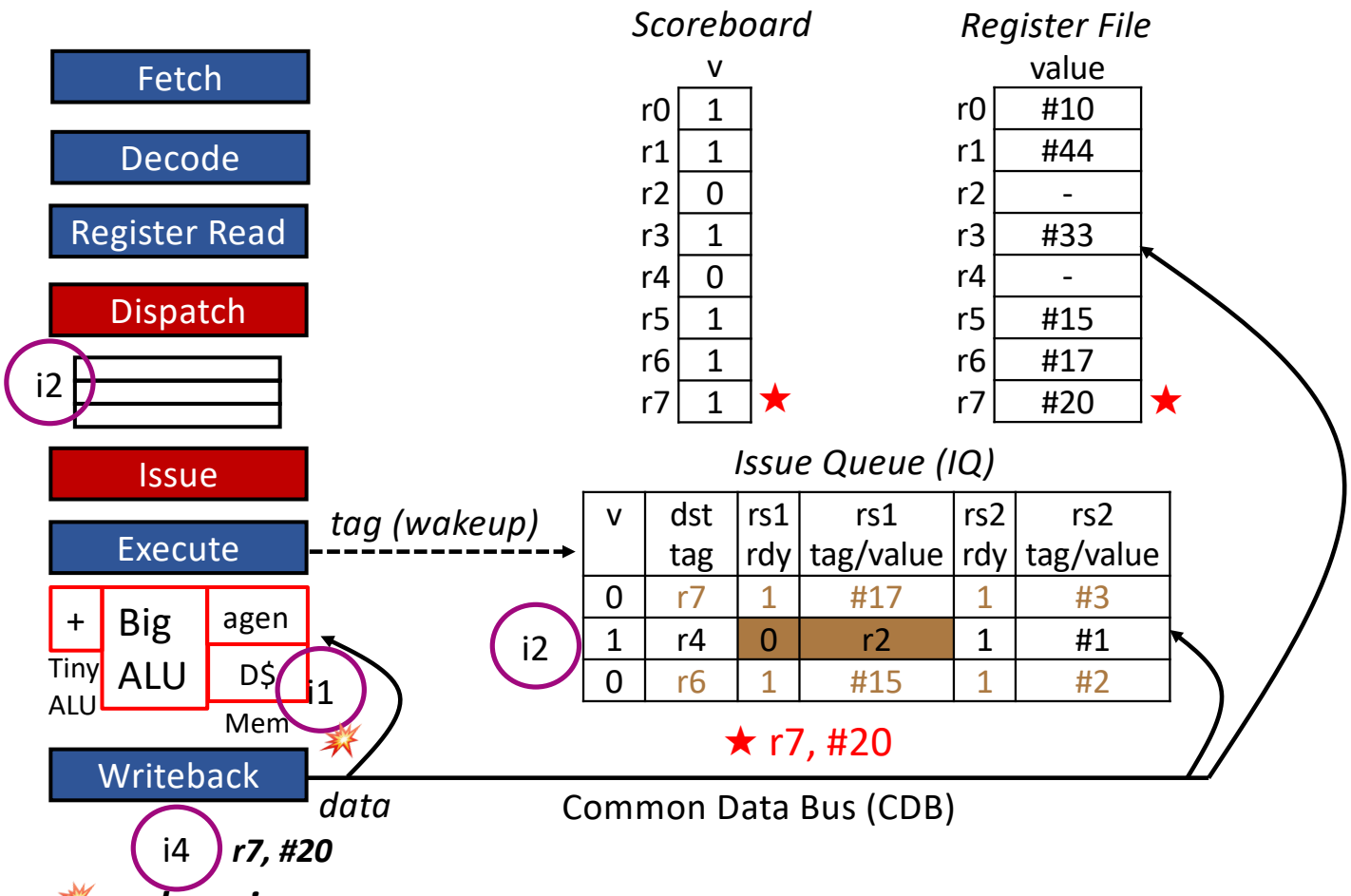
cache miss

r2, @44

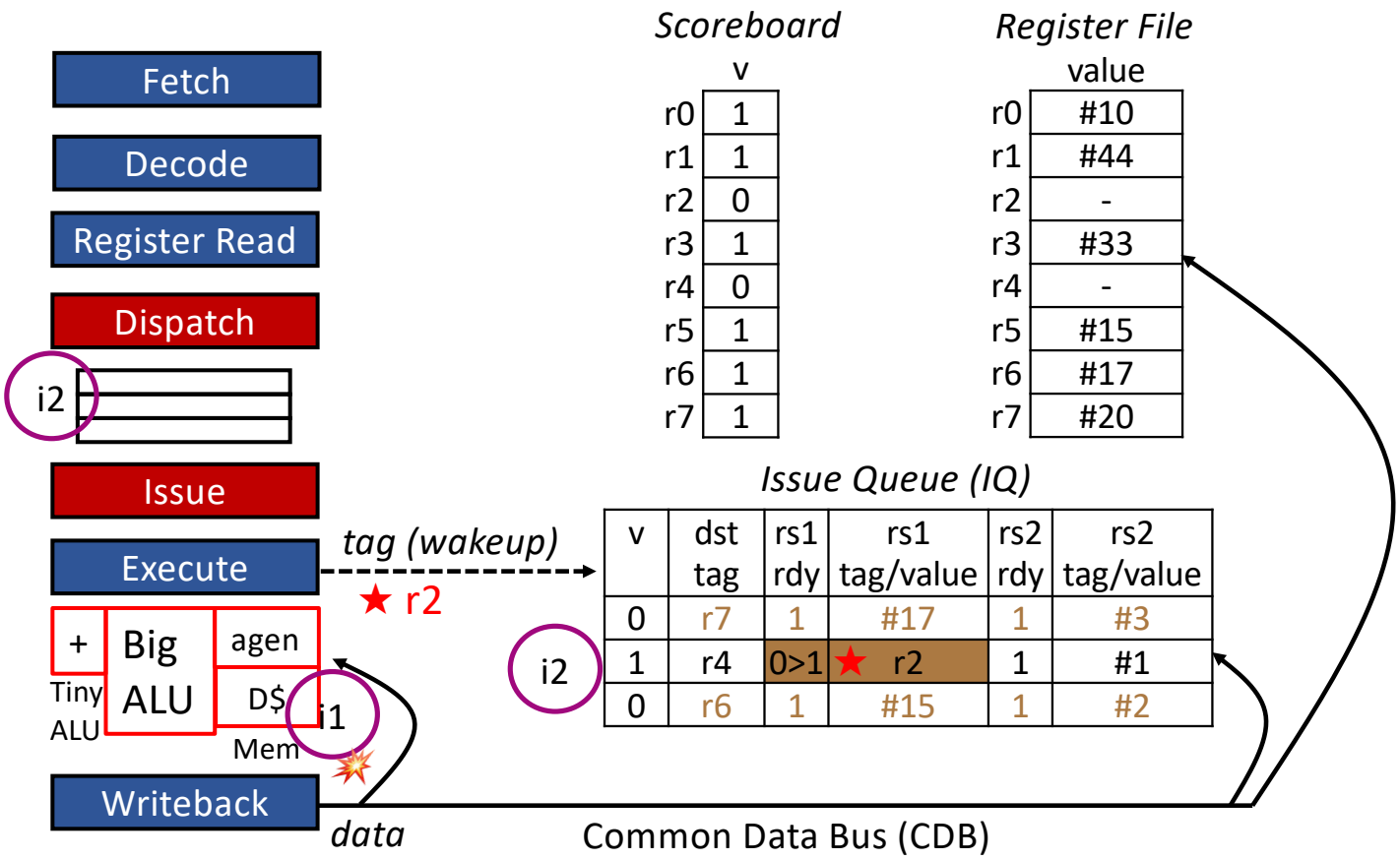
	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
i1: load r2, #0(r1)	FE	DE	RR	DI	IS	$EX_{@}$	$EX_{D\$}$		... miss ...								
i2: add r4, r2, #1		FE	DE	RR	DI	IS	IS	IS									
i3: add r6, r5, #2			FE	DE	RR	DI	IS	EX									
i4: add r7, r6, #3				FE	DE	RR	DI	IS									



	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
i1: load r2, #0(r1)	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								

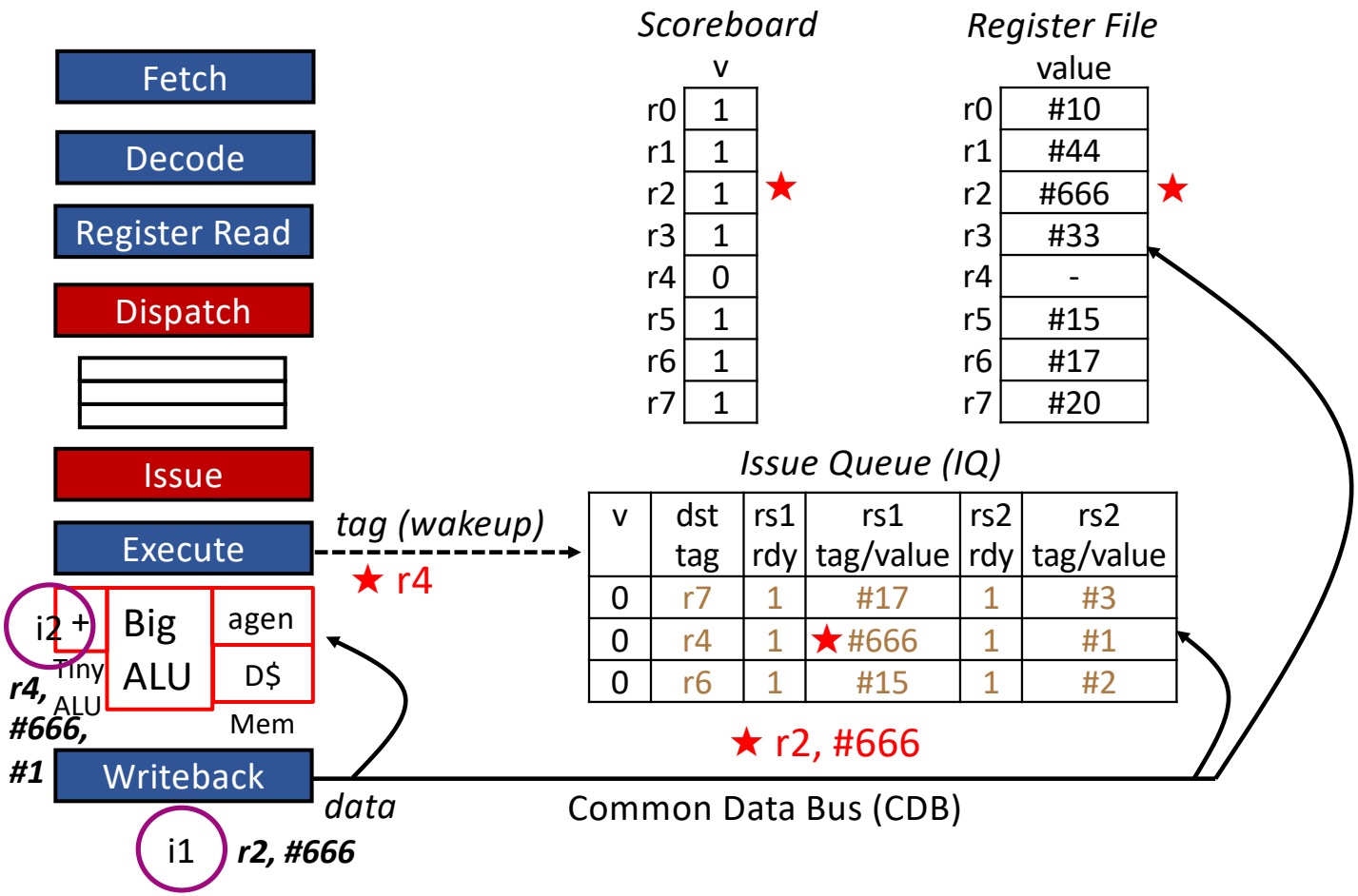


	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
i1: load r2, #0(r1)	FE	DE	RR	DI	IS	$EX_{@}$	$EX_{D\$}$	... miss ...									
i2: add r4, r2, #1		FE	DE	RR	DI	IS	IS	IS	IS	IS							
i3: add r6, r5, #2			FE	DE	RR	DI	IS	EX	WB								
i4: add r7, r6, #3				FE	DE	RR	DI	IS	EX	WB							

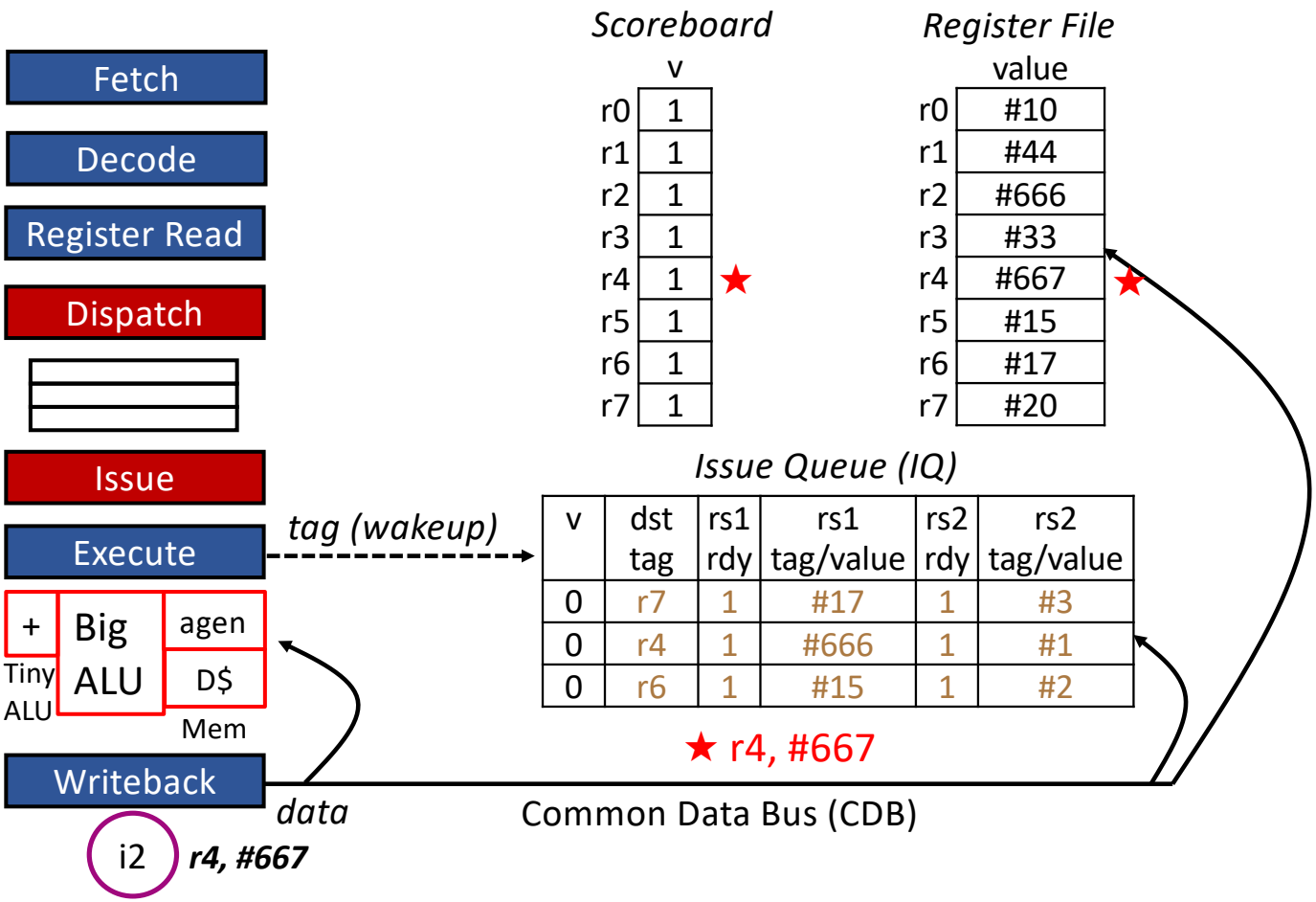


cache miss r2, @44

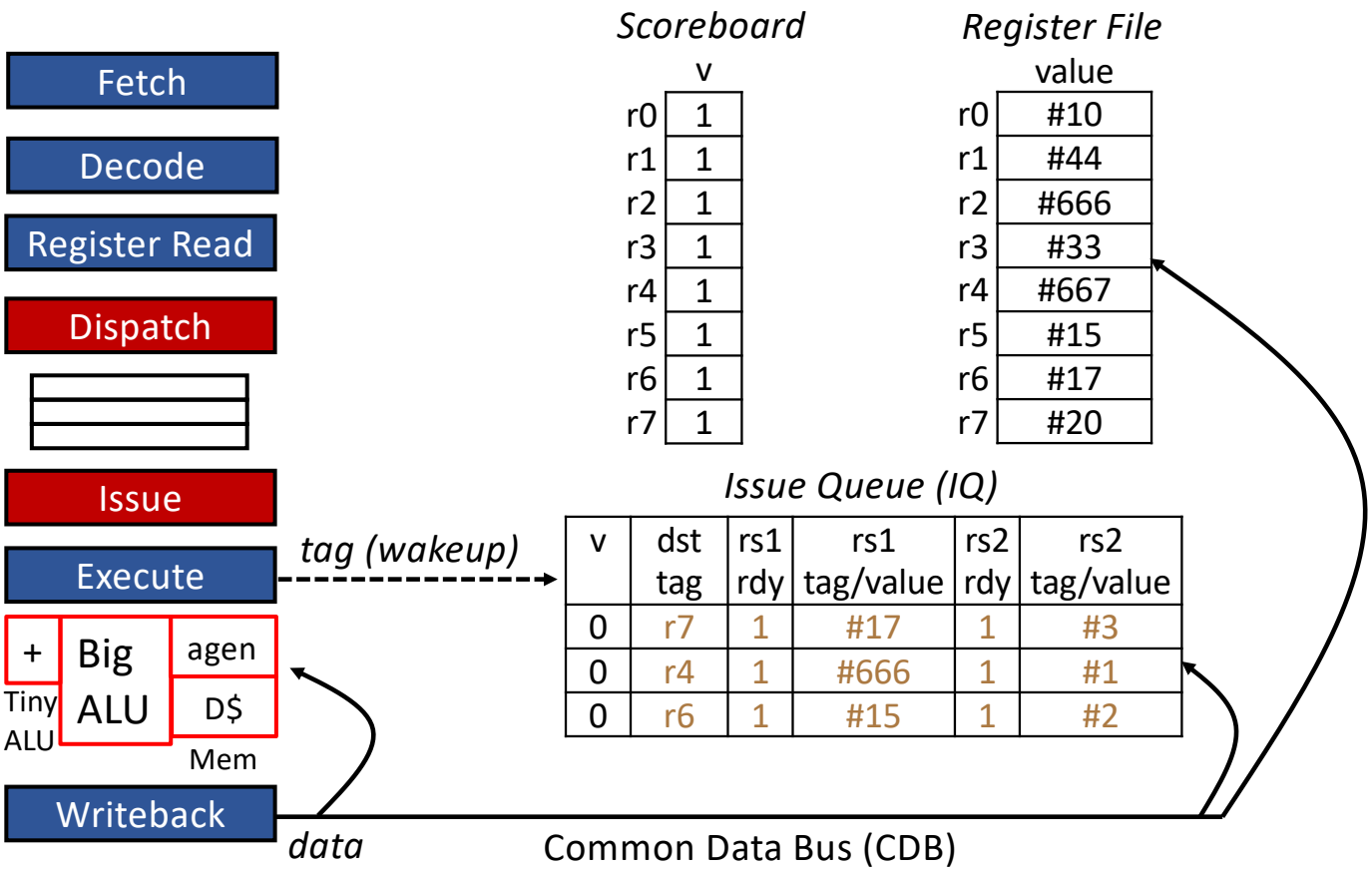
	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
i1: load r2, #0(r1)	FE	DE	RR	DI	IS	$EX_{@}$	$EX_{D\$}$	... miss ...									
i2: add r4, r2, #1		FE	DE	RR	DI	IS	IS	IS	IS	IS	IS						
i3: add r6, r5, #2			FE	DE	RR	DI	IS	EX	WB								
i4: add r7, r6, #3				FE	DE	RR	DI	IS	EX	WB							



	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
i1: load r2, #0(r1)	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: load r2, #0(r1)	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							



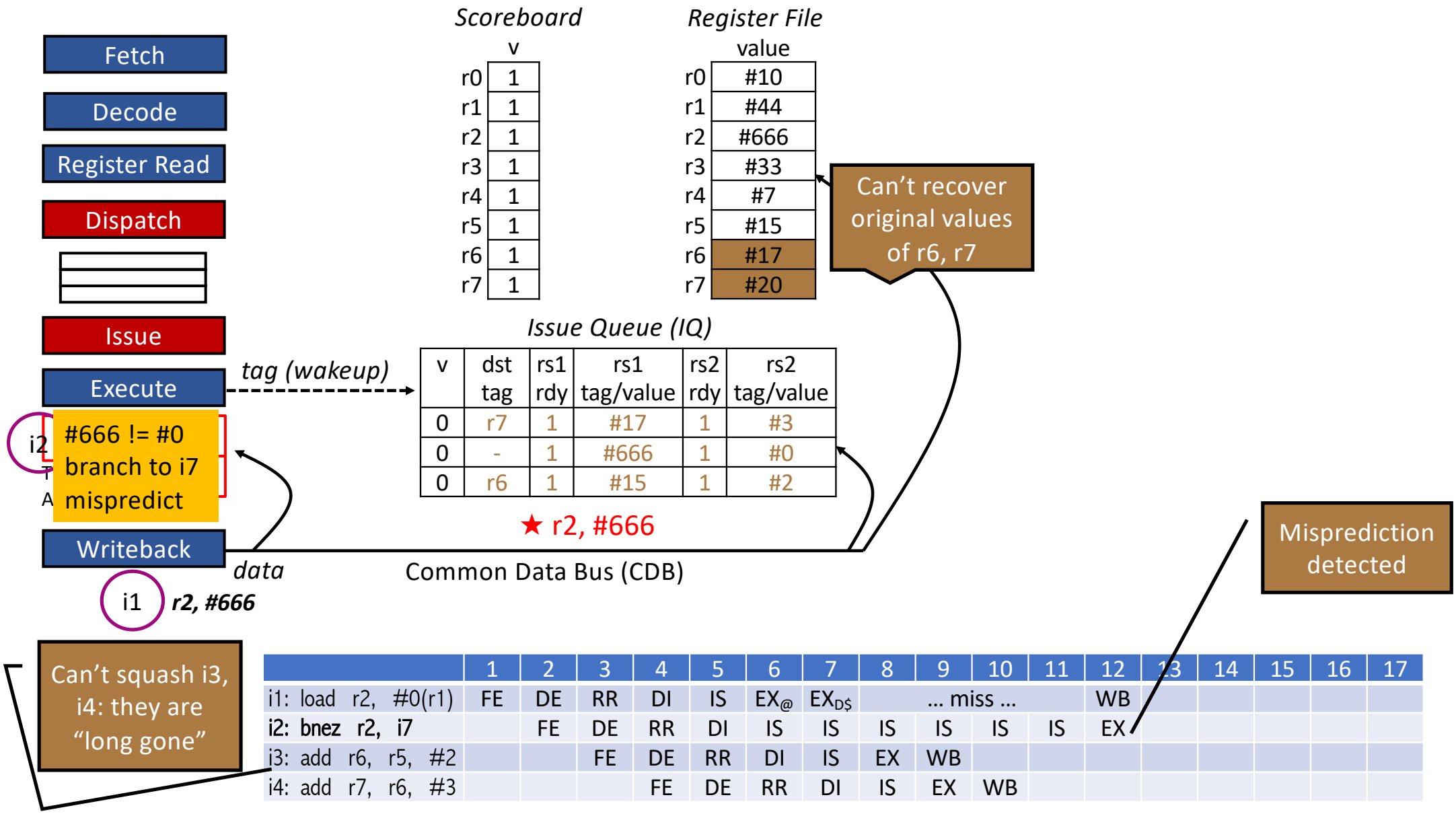
	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
i1: load r2, #0(r1)	FE	DE	RR	DI	IS	$EX_{@}$	$EX_{D\$}$	... miss ...				WB					
i2: add r4, r2, #1		FE	DE	RR	DI	IS	IS	IS	IS	IS	IS	EX	WB				
i3: add r6, r5, #2			FE	DE	RR	DI	IS	EX	WB								
i4: add r7, r6, #3				FE	DE	RR	DI	IS	EX	WB							

Two problems with OOO v.1

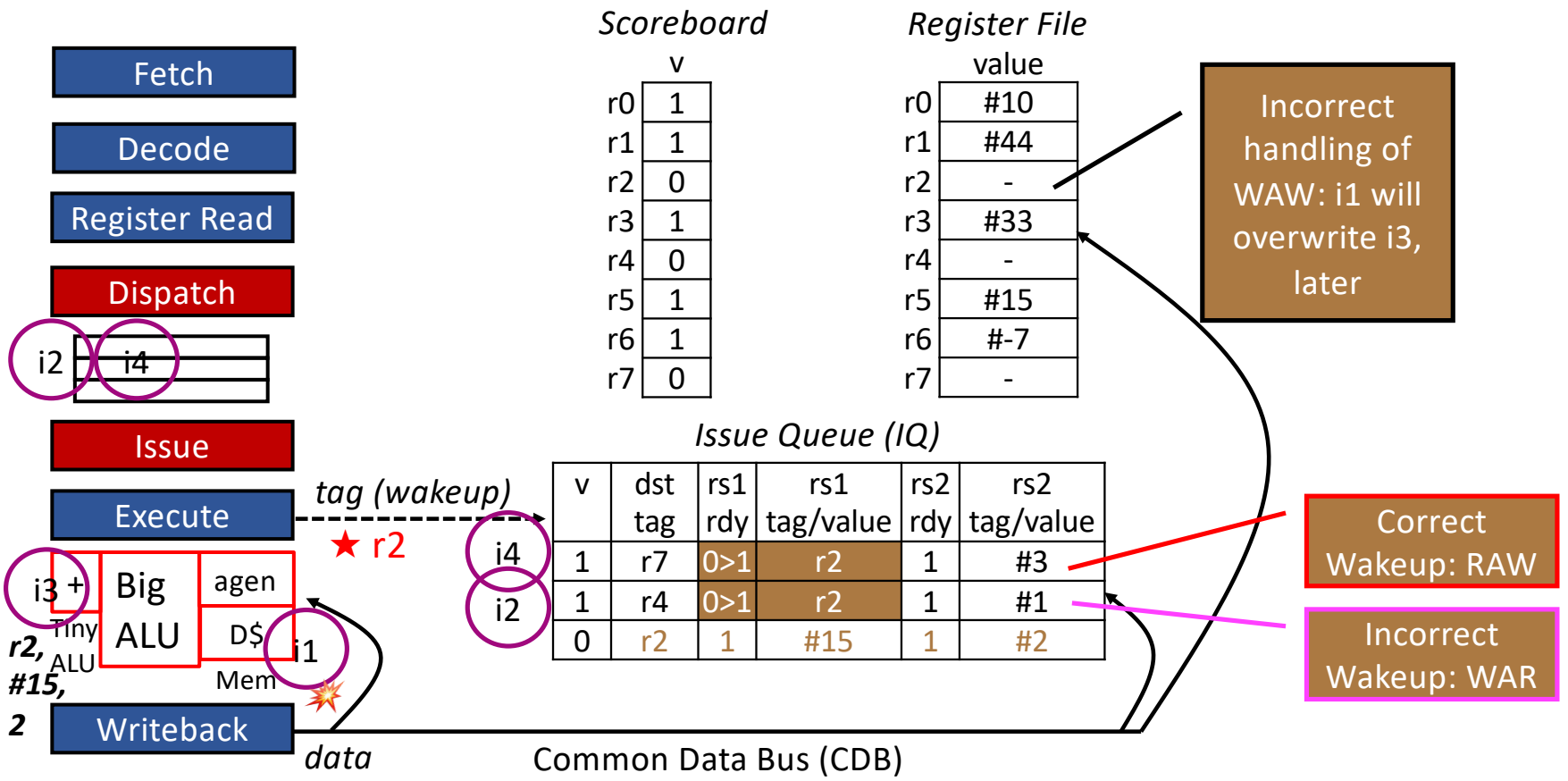
- Cannot recover from misspeculation (cannot schedule past a basic block)
  - Younger instructions are **speculative** with respect to older instructions
  - Possible to have **older predicted branches** that have not executed yet
  - Older load instructions may have executed speculatively with respect 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



	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
i1: load r2, #0(r1)	FE	DE	RR	DI	IS	$EX_{@}$	$EX_{D\$}$	... miss ...				WB					
i2: bnez 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: load r2, #0(r1)	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									

Dynamic Branch Prediction

Dynamic Branch Prediction





- Fork in the road in all cases
  - But, context is different
- **What will you do to go at full-speed?**
  - **Static prediction:** always-left, always-right, return if wrong
  - **Dynamic prediction:** memorize <context-direction> pair in head: <cherry tree, right>, <windmills, left>

Dynamic Branch Prediction

- Predict the outcome of a branch instruction (in fetch stage) based on the recent behavior of the branch

- **What do we need?**
  - Branch identification (`PC` uniquely identifies a branch)
  - Recent branch behavior (taken/untaken last time)

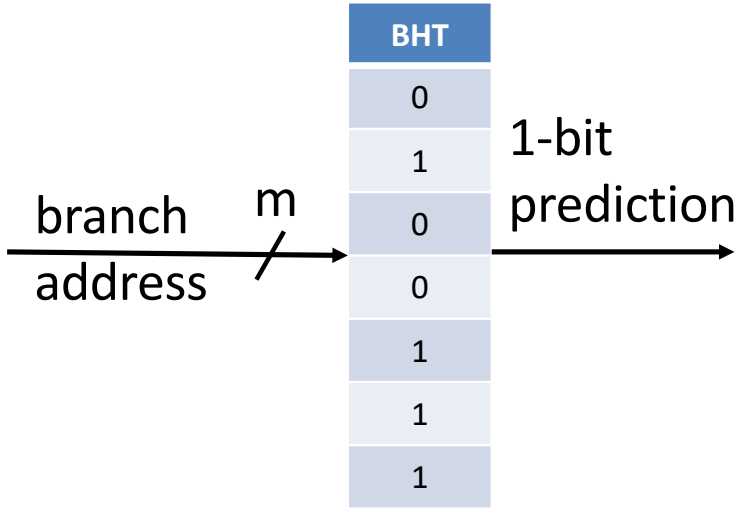
Branch Identification & Behavior

- **Branch identification**
  - Use the branch address in instruction memory
  - Can grab it from `PC`

- **Branch behavior**
  - Outcome of the **condition test** from ALU
  - Can also store the **branch target** the last time the branch executed

One-Bit Predictor

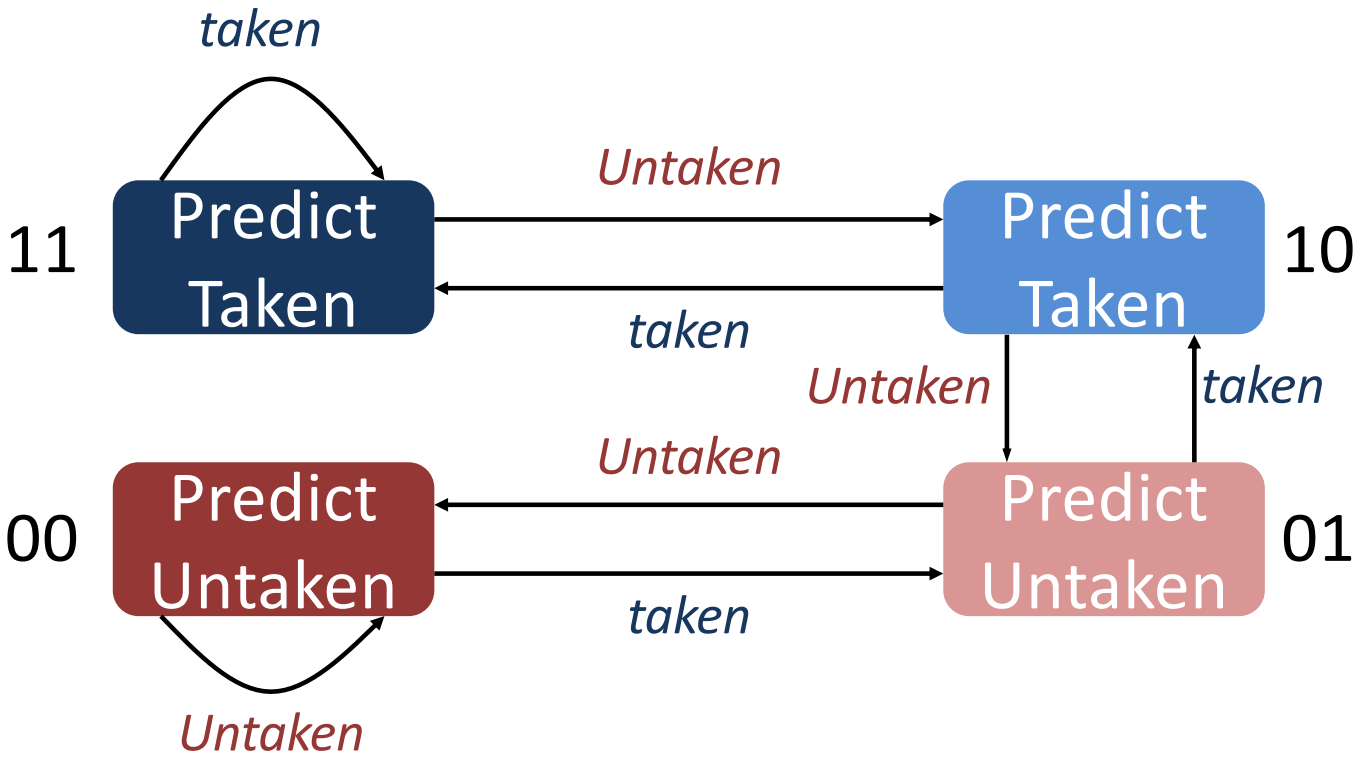
- Branch History Table (BHT) or Branch Prediction Buffer
  - *A small amount of memory indexed by the low-order bits of branch address*
- **Key Idea:** *Store a single bit that says branch was recently taken or not*



Due to limited entries in the table, there are conflicts (aka. aliasing)

Smith Predictor

The state transitions show the *bimodal* behavior of Smith predictor



Global Branch Correlation

- Can we predict the behavior of one branch based on the direction of another branch?

```
if (aa == 2)          B1
        aa = 0;
if (bb == 2)          B2
        bb = 0
if (aa != bb)         B3
        {…}
```

(B1, B2, B3)
(T, T, ?)
(T, F, ?)

```
if (counter > 15)                 B1
{
        reset = 1;
}
…
…
if (reset == 1)                   B2
        {…}
if (counter < 2)
        {…}                       B3
```

(B1, B2, B3)
(T, ?, ?)
(F, ?, ?)

A Lot More to Say on Branch Prediction!

- Important component of a modern processor
  - Especially superscalar and out-of-order processors
- We correlating local/global history predictors as well
  - Branches in programs are correlated
- Prediction accuracy above 90%
- **State of art:** Deep neural networks, machine learning approaches
- **Random branches** are increasingly common (ML, NLP, GPT)

Two Humps in a Modern Pipeline

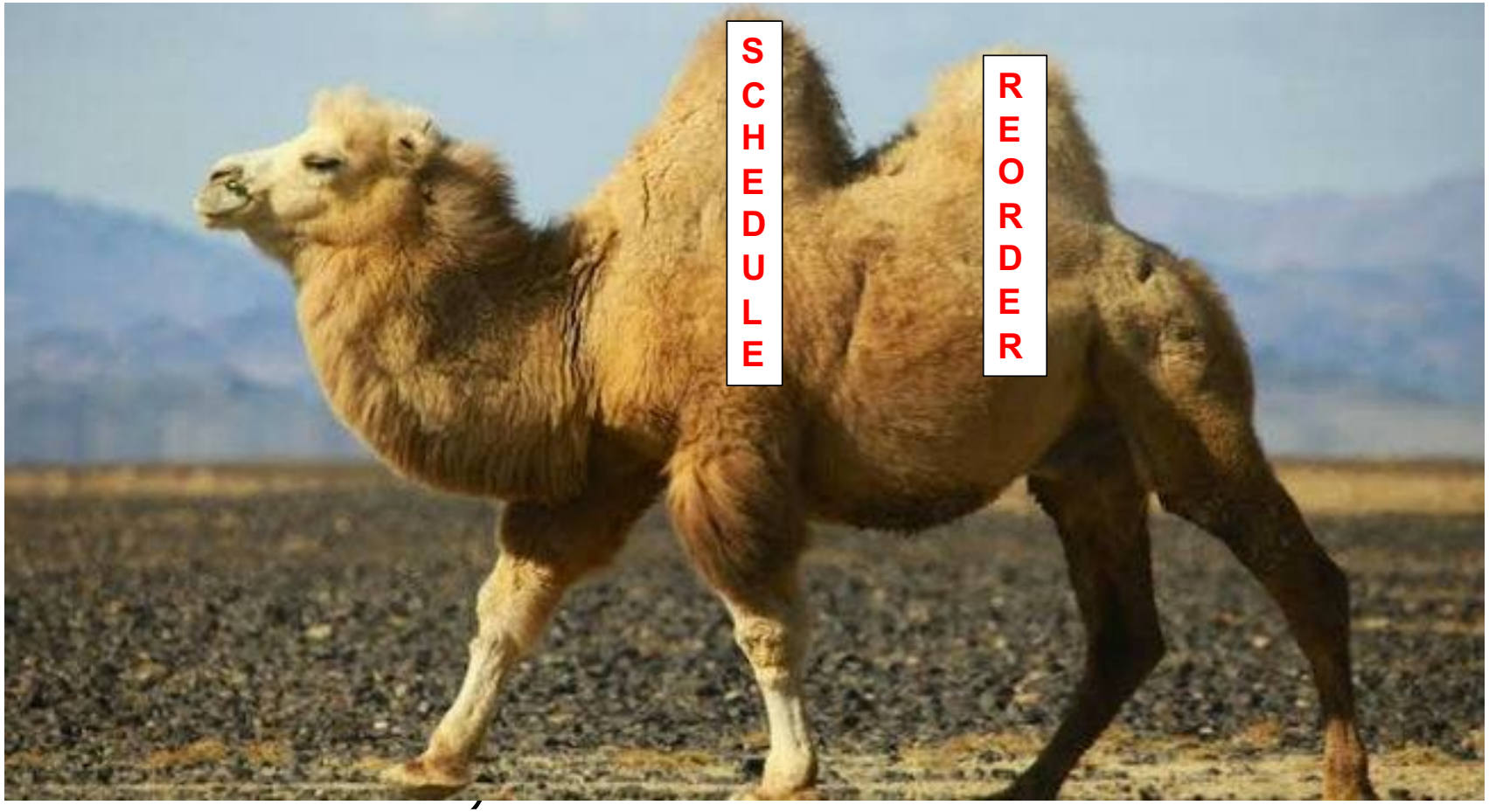


Photo credit: http://true-wildlife.blogspot.ch/2010/10/bactrian-camel.html

Hardware Speculation

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

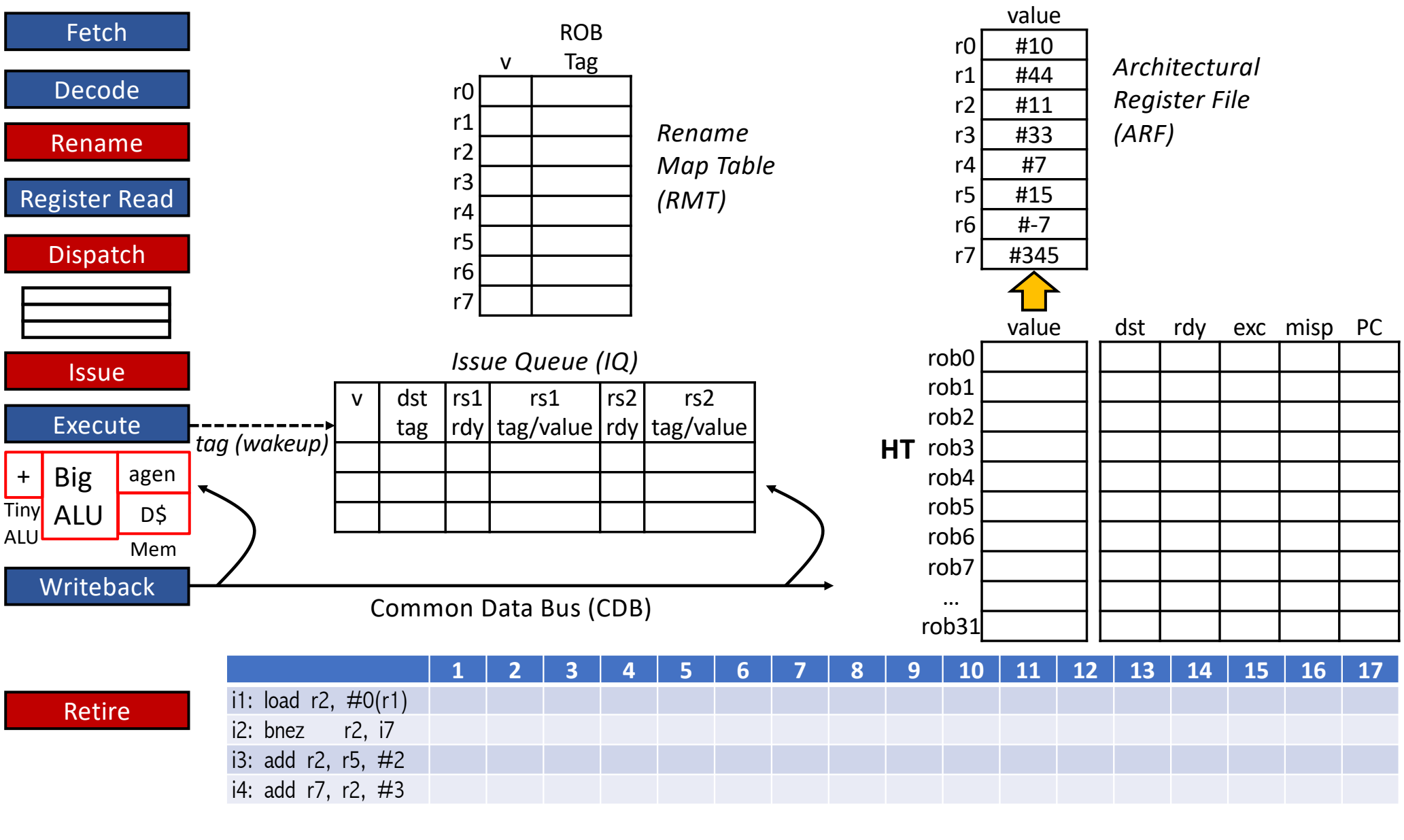
- **Register Read Stage**
  - Obtain source value from ARF or ROB (using renamed source)
  - If renamed to ROB, ROB may indicate value not ready yet
    - Producer hasn't executed yet
    - Keep renamed source as proxy for value
  - A consumer instruction obtains its source values from ARF, ROB, and/or 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

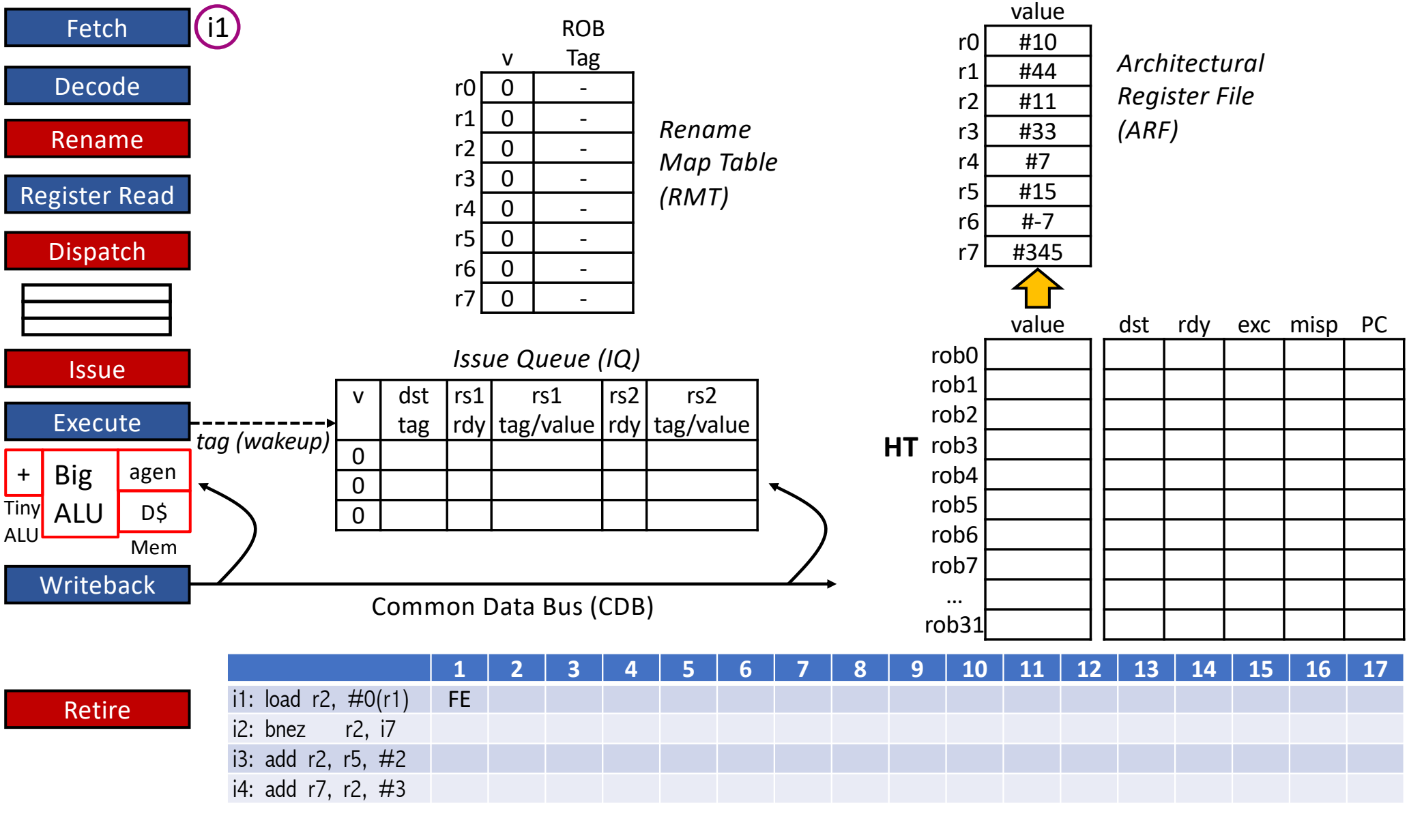
- Writeback Stage
  - Instruction writes its speculative result OOO into ROB instead of ARF (at its ROB entry)

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

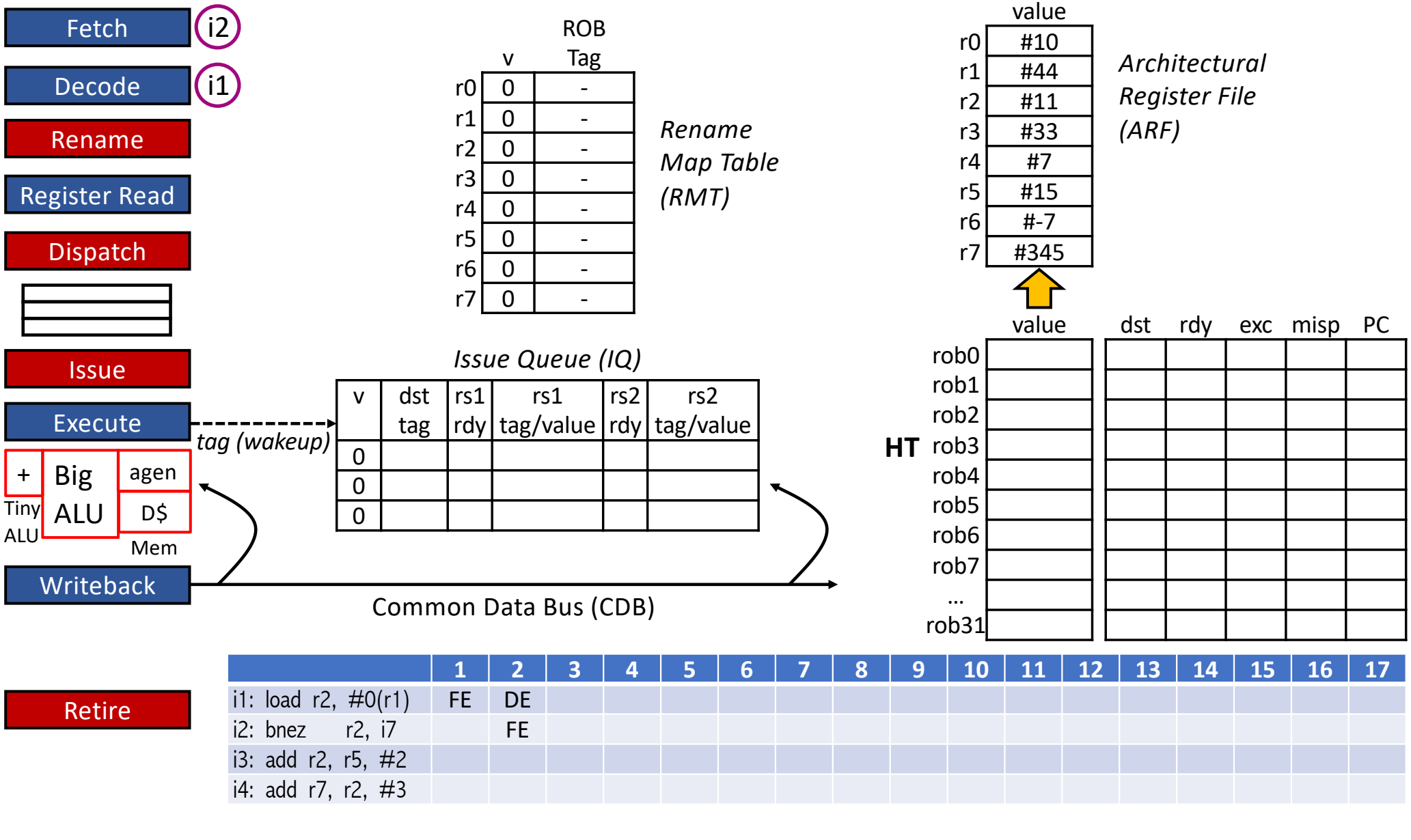
- Misprediction/exception recovery
  - Offending instruction posts misprediction or exception bit in its ROB entry OOO
  - Wait until offending instruction reaches head of ROB (oldest unretired instruction)
  - **Squash all instructions** in pipeline and ROB, and restore RMT to be consistent with an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



	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
i1: load r2, #0(r1)																	
i2: bnez r2, i7																	
i3: add r2, r5, #2																	
i4: add r7, r2, #3																	

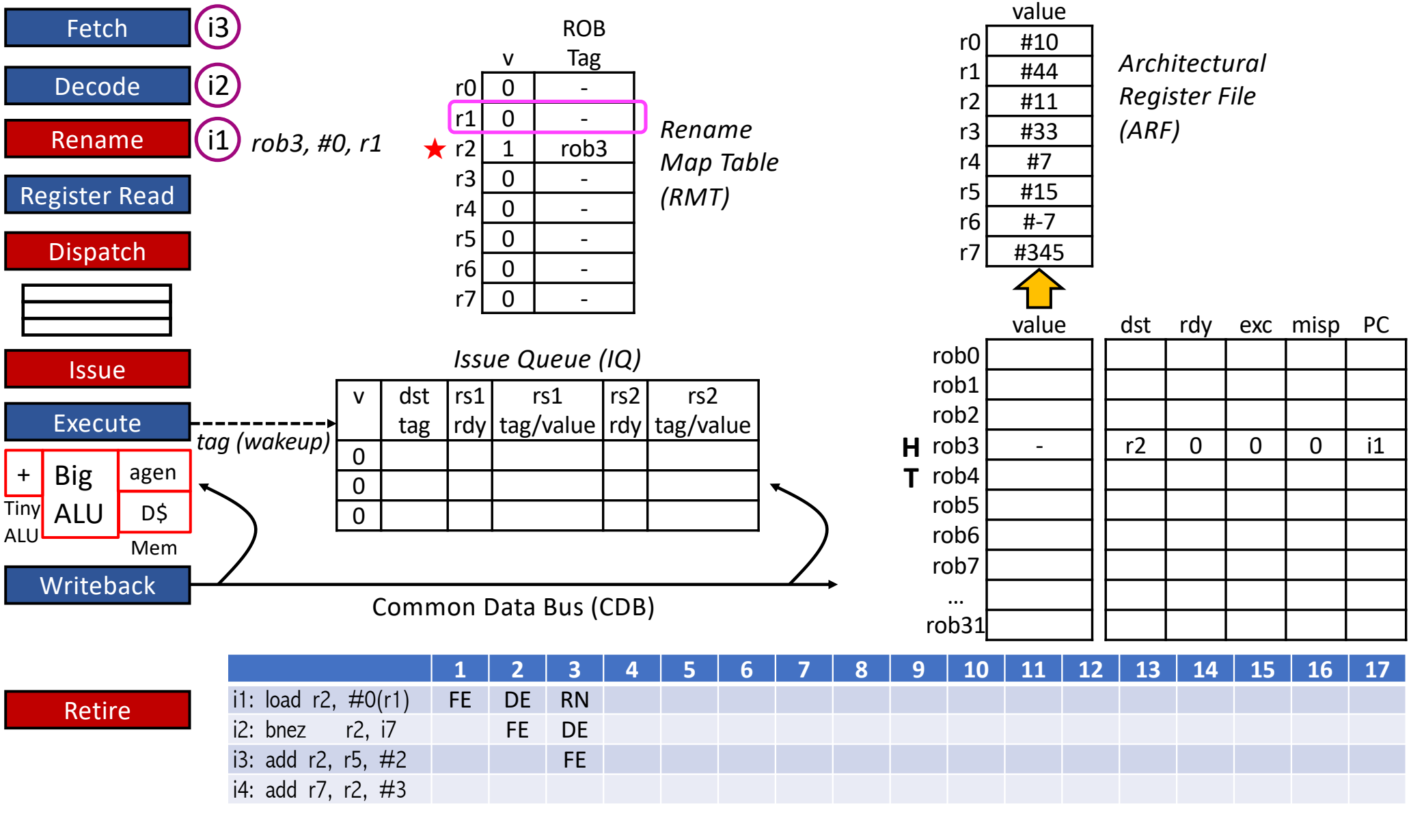


	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
i1: load r2, #0(r1)	FE																
i2: bnez r2, i7																	
i3: add r2, r5, #2																	
i4: add r7, r2, #3																	

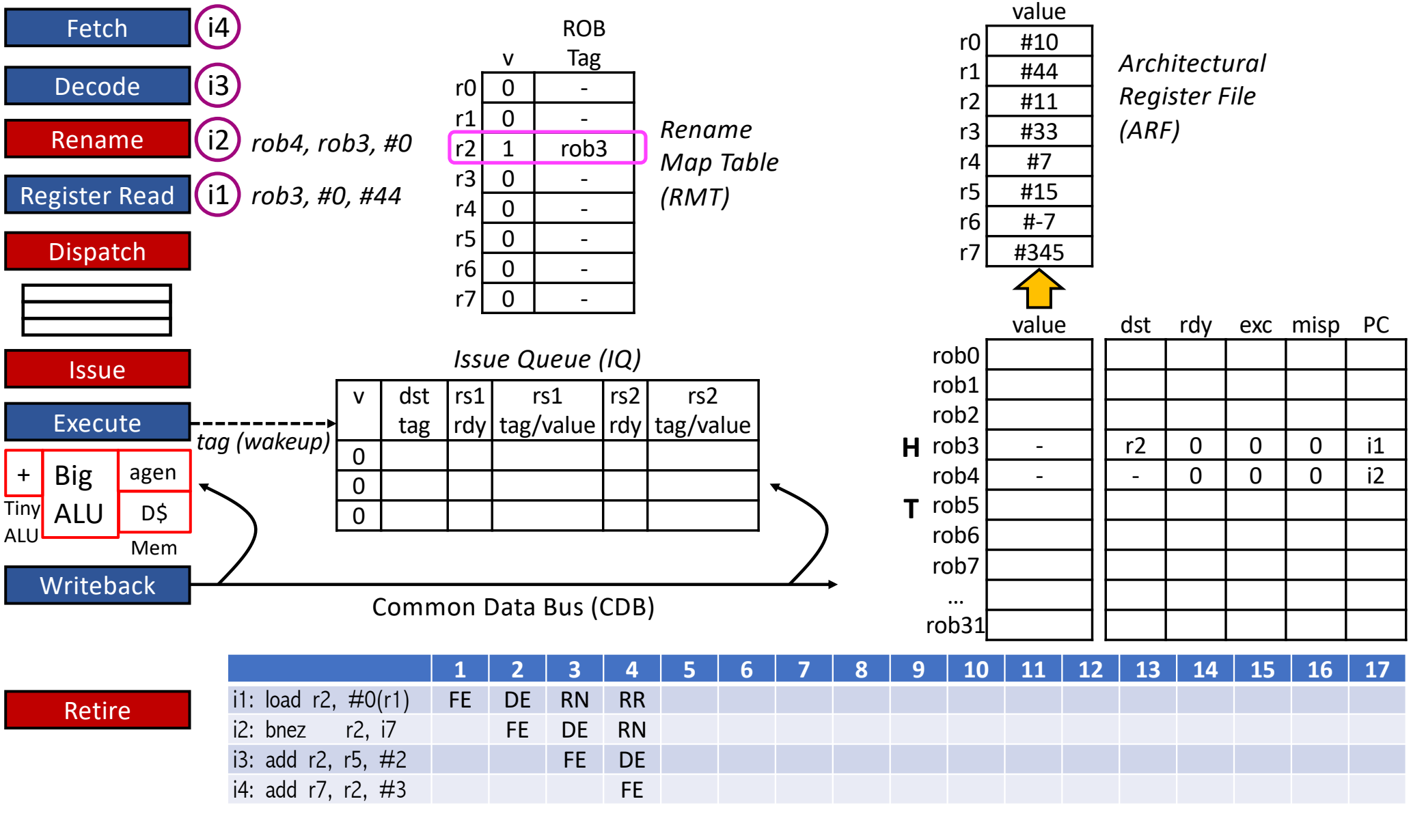


Retire

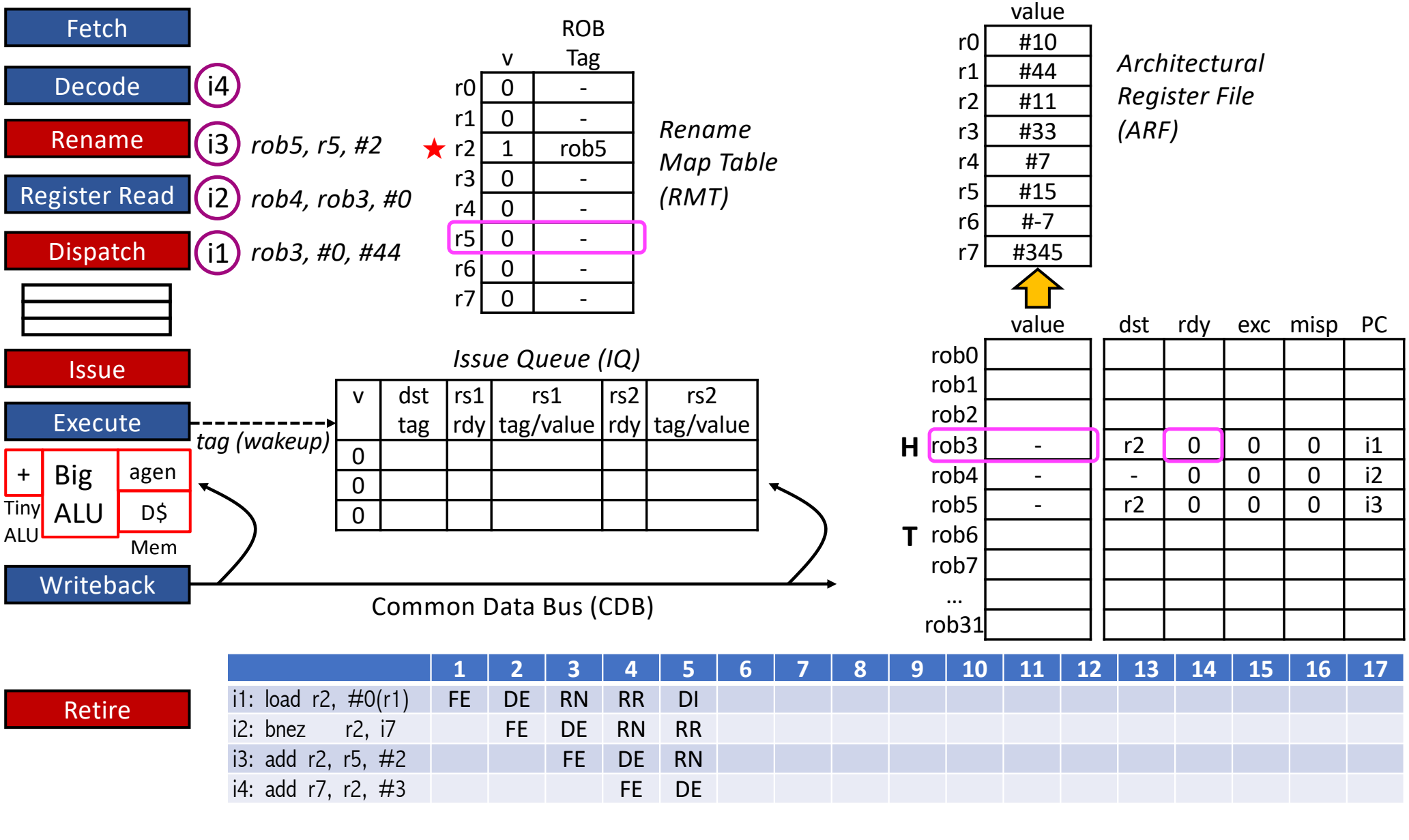
	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
i1: load r2, #0(r1)	FE	DE															
i2: bnez r2, i7		FE															
i3: add r2, r5, #2																	
i4: add r7, r2, #3																	



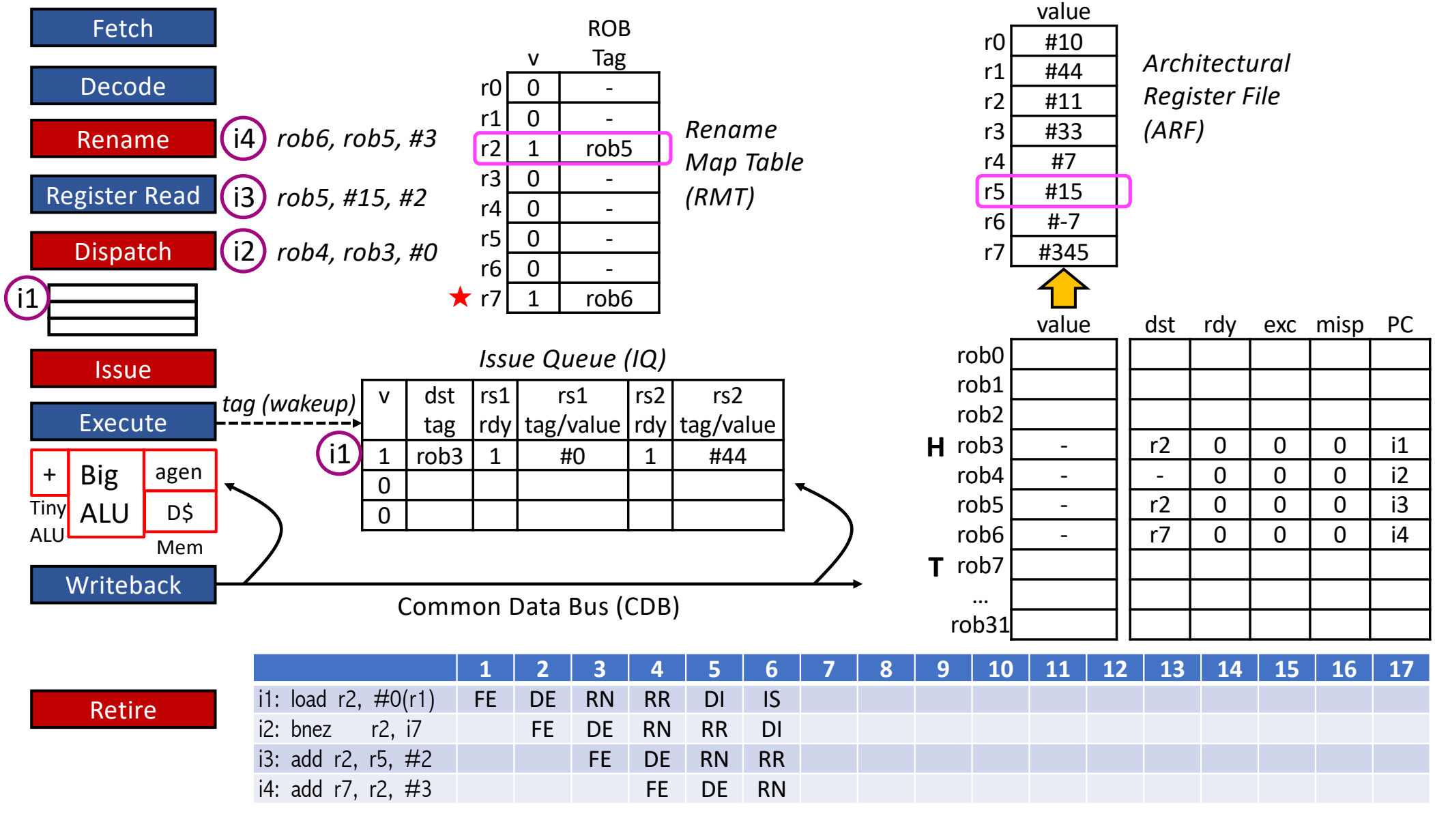
	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
i1: load r2, #0(r1)	FE	DE	RN														
i2: bnez r2, i7		FE	DE														
i3: add r2, r5, #2			FE														
i4: add r7, r2, #3																	



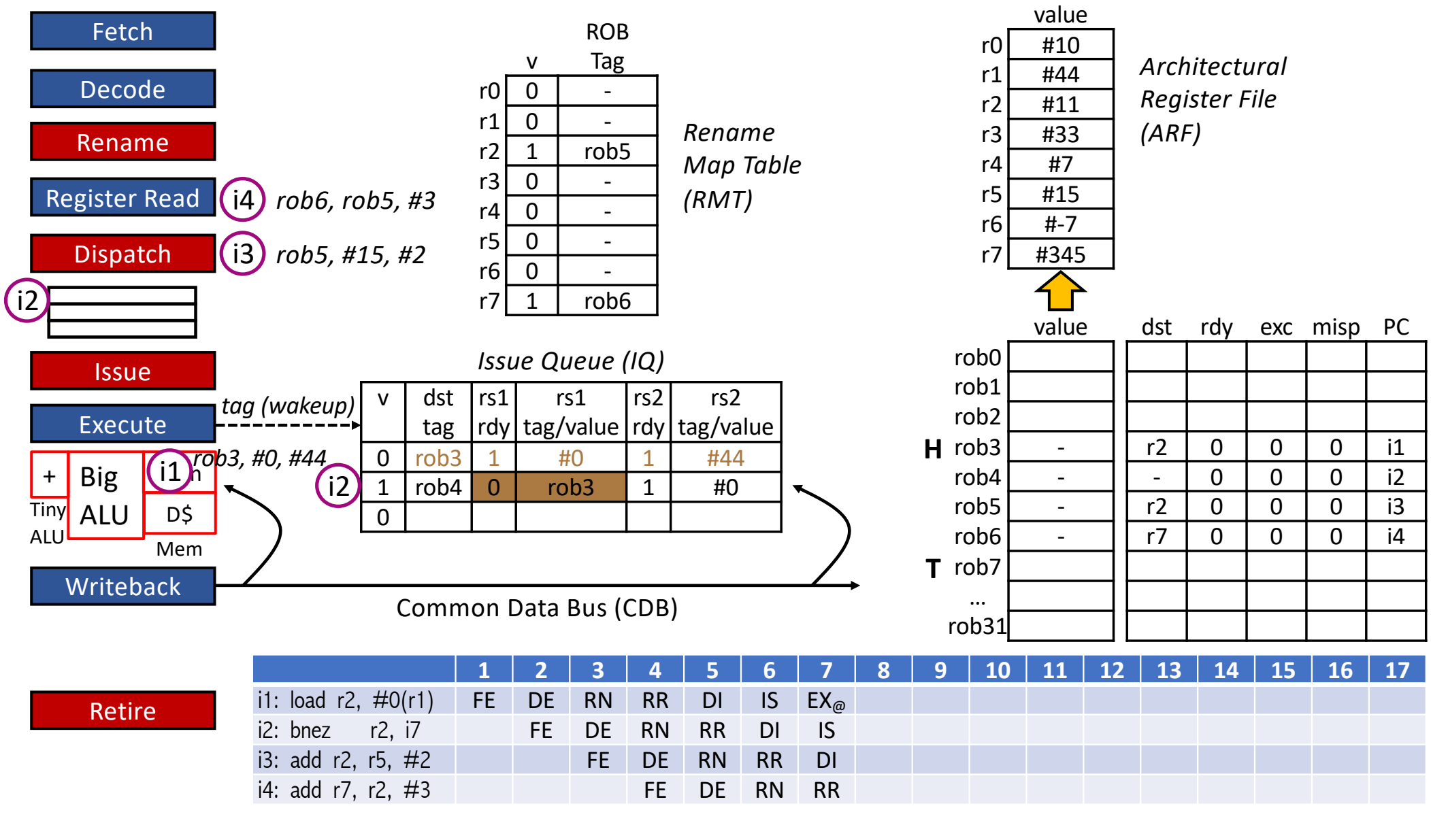
	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
i1: load r2, #0(r1)	FE	DE	RN	RR													
i2: bnez r2, i7		FE	DE	RN													
i3: add r2, r5, #2			FE	DE													
i4: add r7, r2, #3				FE													



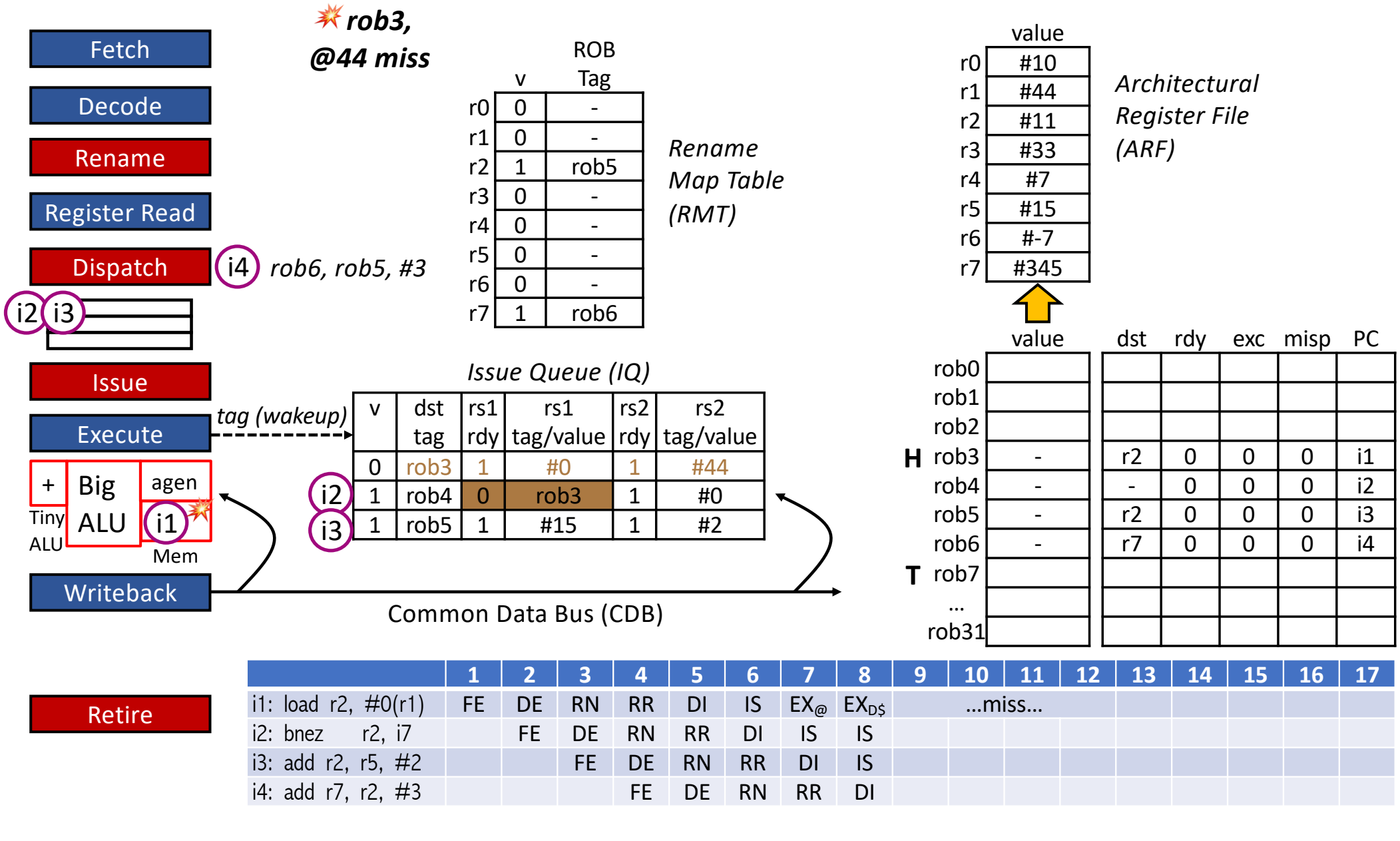
	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
i1: load r2, #0(r1)	FE	DE	RN	RR	DI												
i2: bnez r2, i7		FE	DE	RN	RR												
i3: add r2, r5, #2			FE	DE	RN												
i4: add r7, r2, #3				FE	DE												



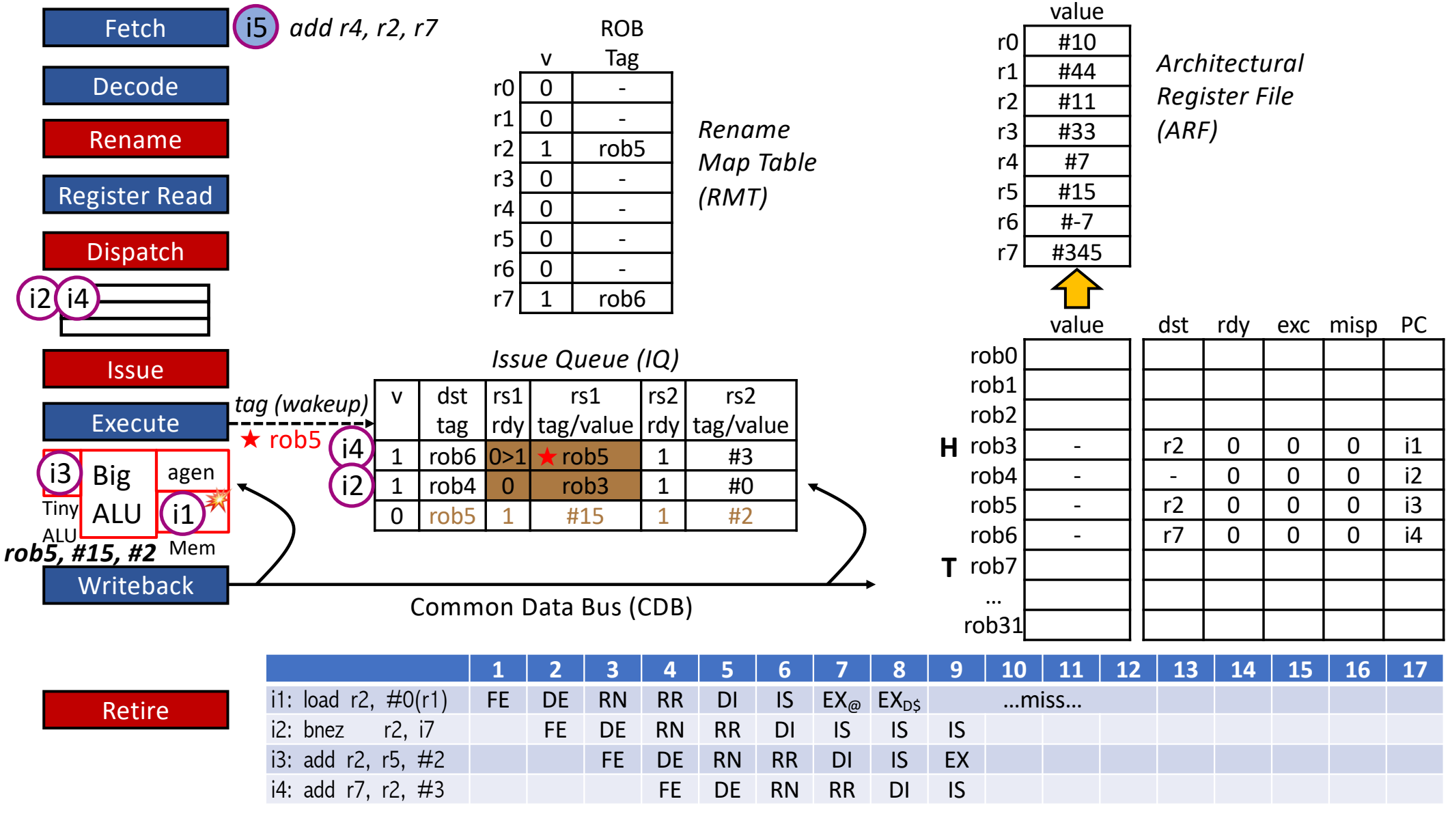
	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
i1: load r2, #0(r1)	FE	DE	RN	RR	DI	IS											
i2: bnez r2, i7		FE	DE	RN	RR	DI											
i3: add r2, r5, #2			FE	DE	RN	RR											
i4: add r7, r2, #3				FE	DE	RN											



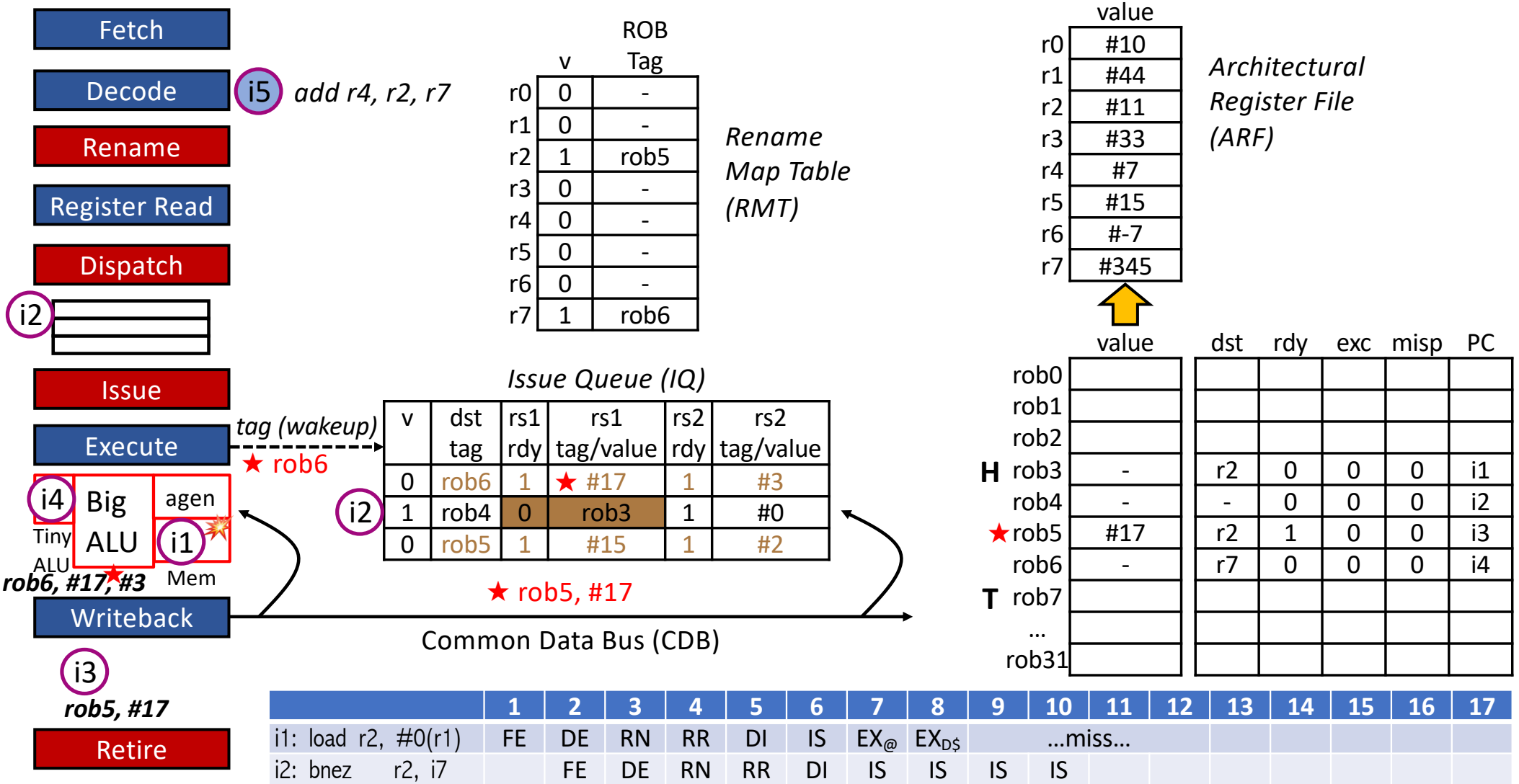
	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
i1: load r2, #0(r1)	FE	DE	RN	RR	DI	IS	$EX_{@}$										
i2: bnez r2, i7		FE	DE	RN	RR	DI	IS										
i3: add r2, r5, #2			FE	DE	RN	RR	DI										
i4: add r7, r2, #3				FE	DE	RN	RR										



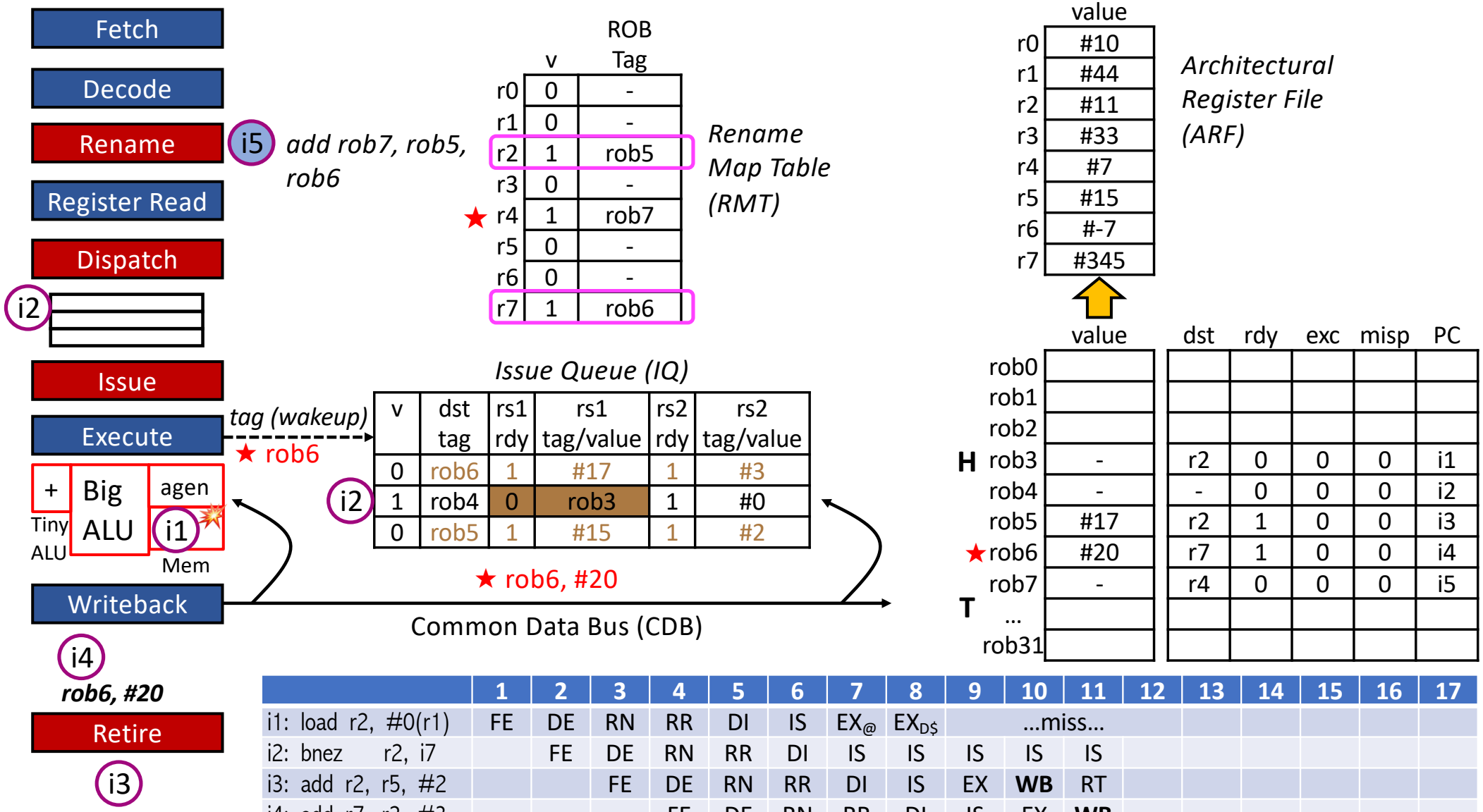
	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
i1: load r2, #0(r1)	FE	DE	RN	RR	DI	IS	$EX_{@}$	$EX_{D\$}$	...miss...								
i2: bnez r2, i7		FE	DE	RN	RR	DI	IS	IS									
i3: add r2, r5, #2			FE	DE	RN	RR	DI	IS									
i4: add r7, r2, #3				FE	DE	RN	RR	DI									



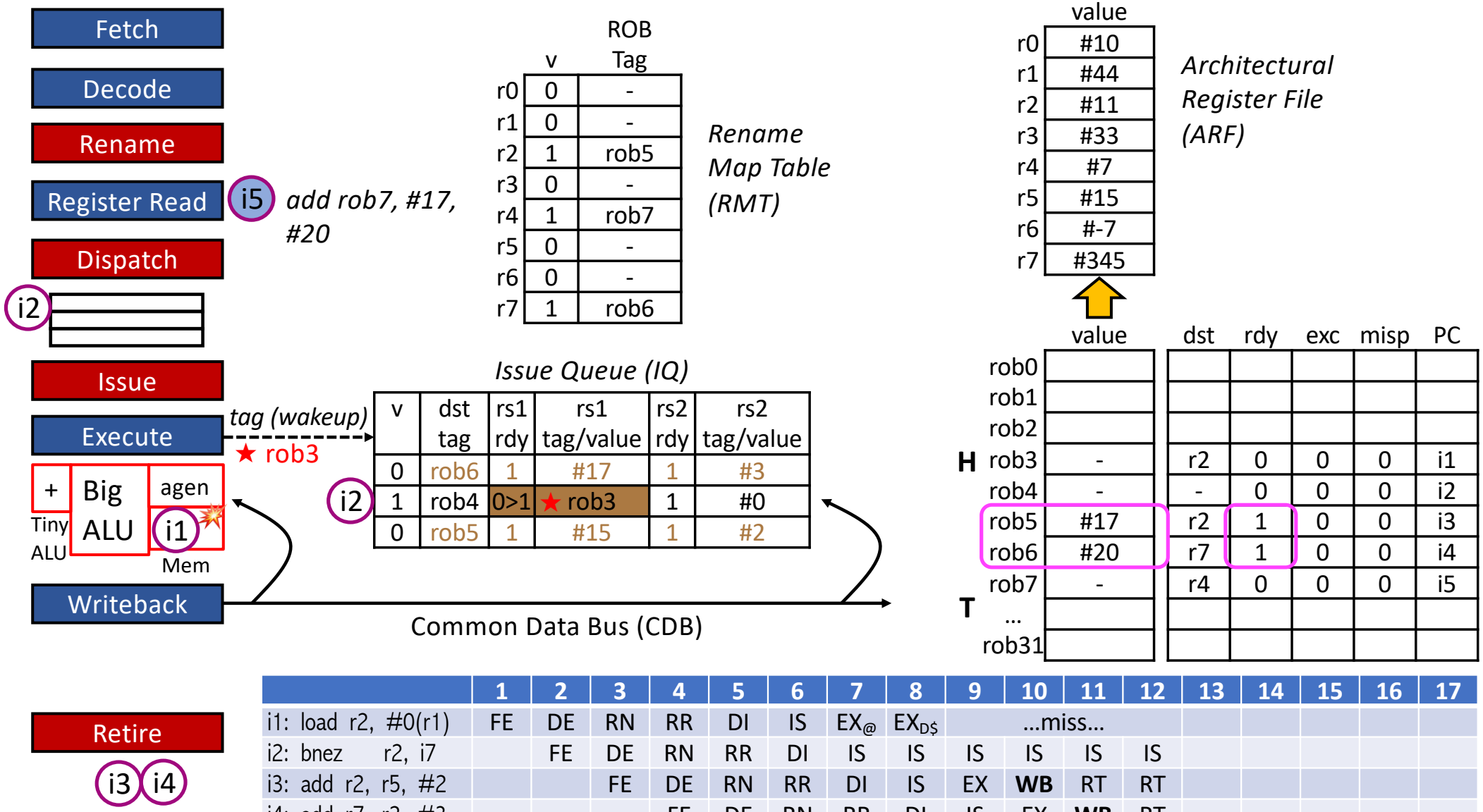
	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
i1: load r2, #0(r1)	FE	DE	RN	RR	DI	IS	$EX_{@}$	$EX_{D\$}$	...miss...								
i2: bnez r2, i7		FE	DE	RN	RR	DI	IS	IS	IS								
i3: add r2, r5, #2			FE	DE	RN	RR	DI	IS	EX								
i4: add r7, r2, #3				FE	DE	RN	RR	DI	IS								



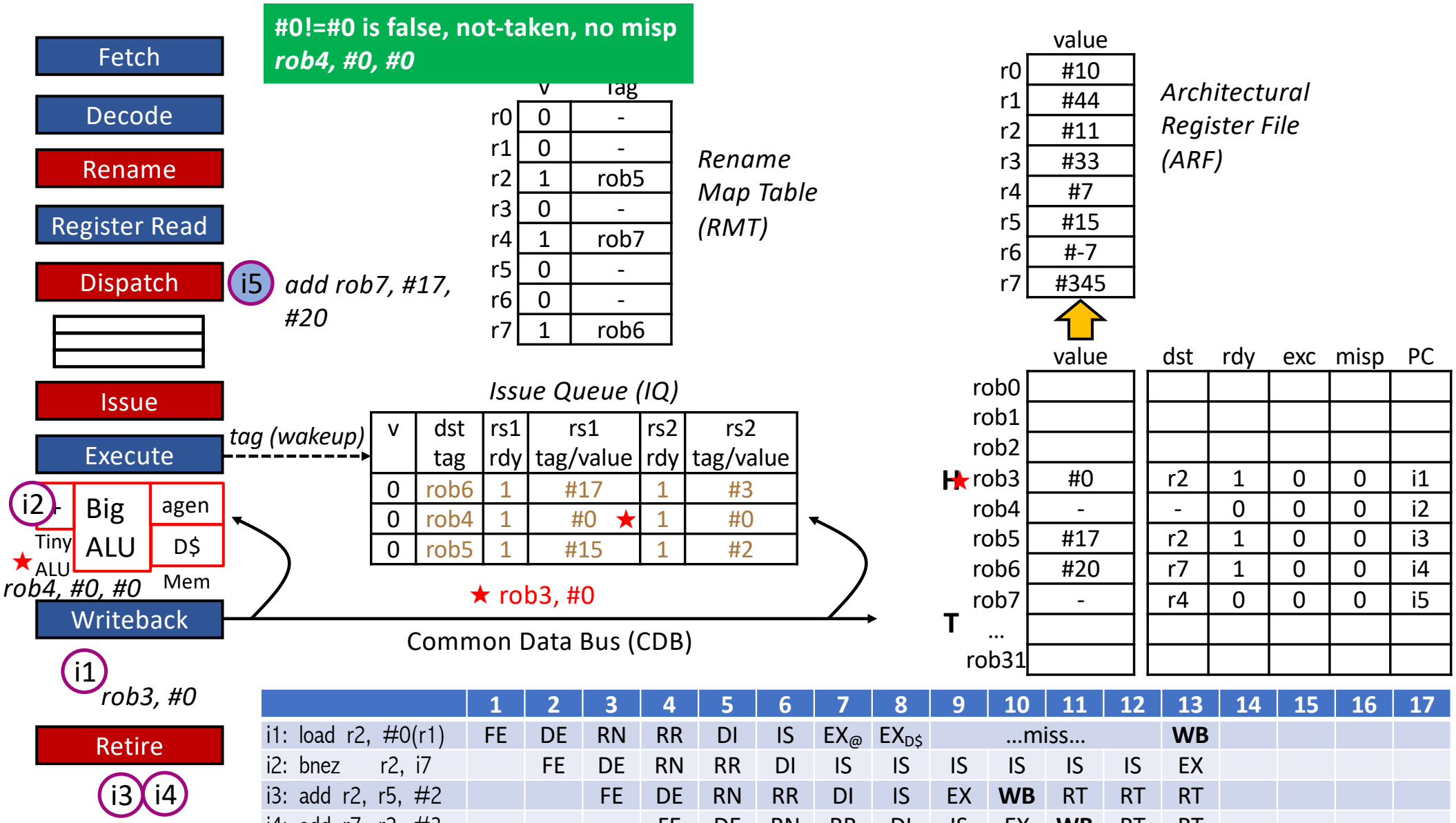
	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
i1: load r2, #0(r1)	FE	DE	RN	RR	DI	IS	$EX_{@}$	$EX_{D\$}$	...miss...								
i2: bnez r2, i7		FE	DE	RN	RR	DI	IS	IS	IS	IS							
i3: add r2, r5, #2			FE	DE	RN	RR	DI	IS	EX	**WB**							
i4: add r7, r2, #3				FE	DE	RN	RR	DI	IS	EX							



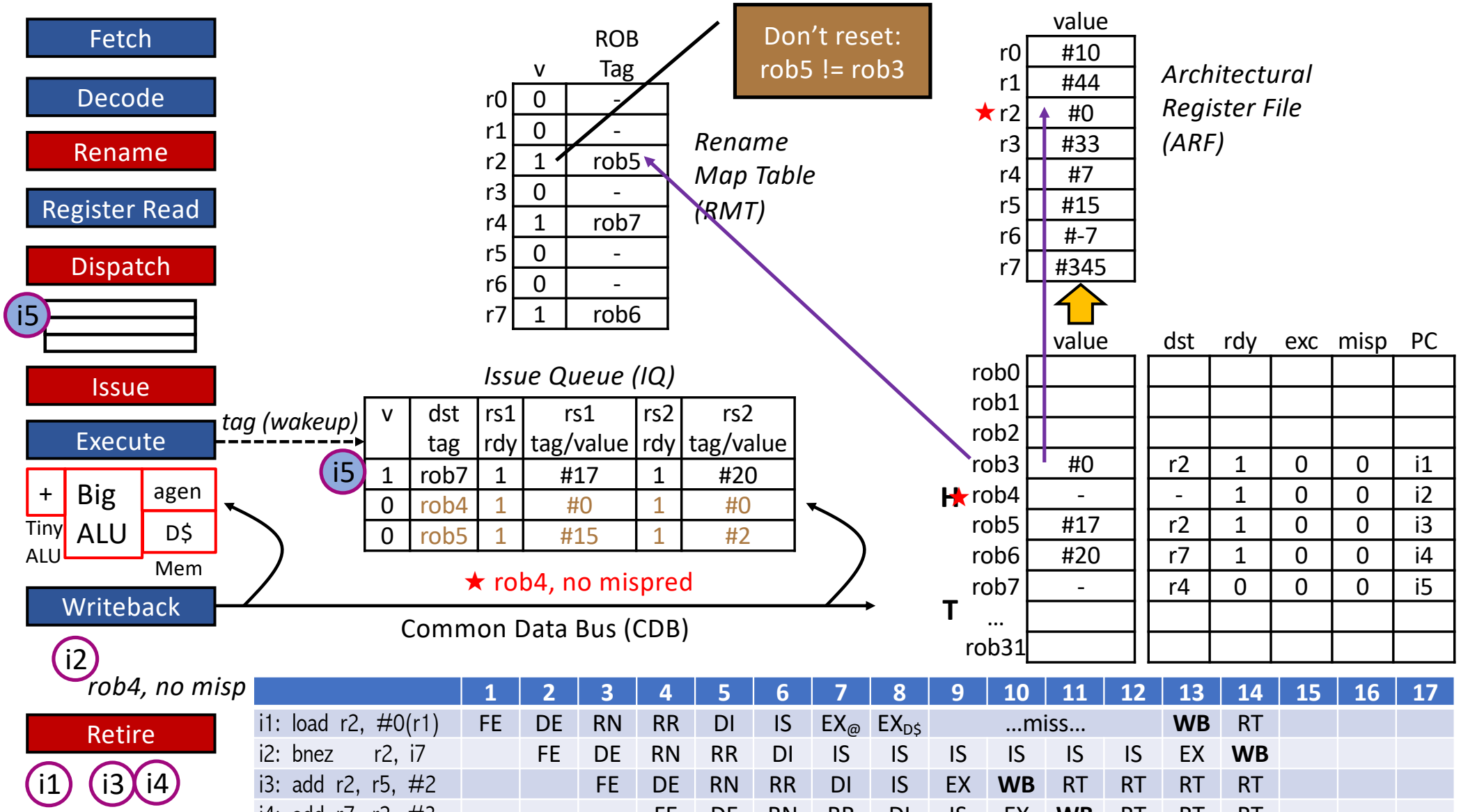
	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
i1: load r2, #0(r1)	FE	DE	RN	RR	DI	IS	$EX_{@}$	$EX_{D\$}$	...miss...								
i2: bnez r2, i7		FE	DE	RN	RR	DI	IS	IS	IS	IS	IS						
i3: add r2, r5, #2			FE	DE	RN	RR	DI	IS	EX	**WB**	RT						
i4: add r7, r2, #3				FE	DE	RN	RR	DI	IS	EX	**WB**						



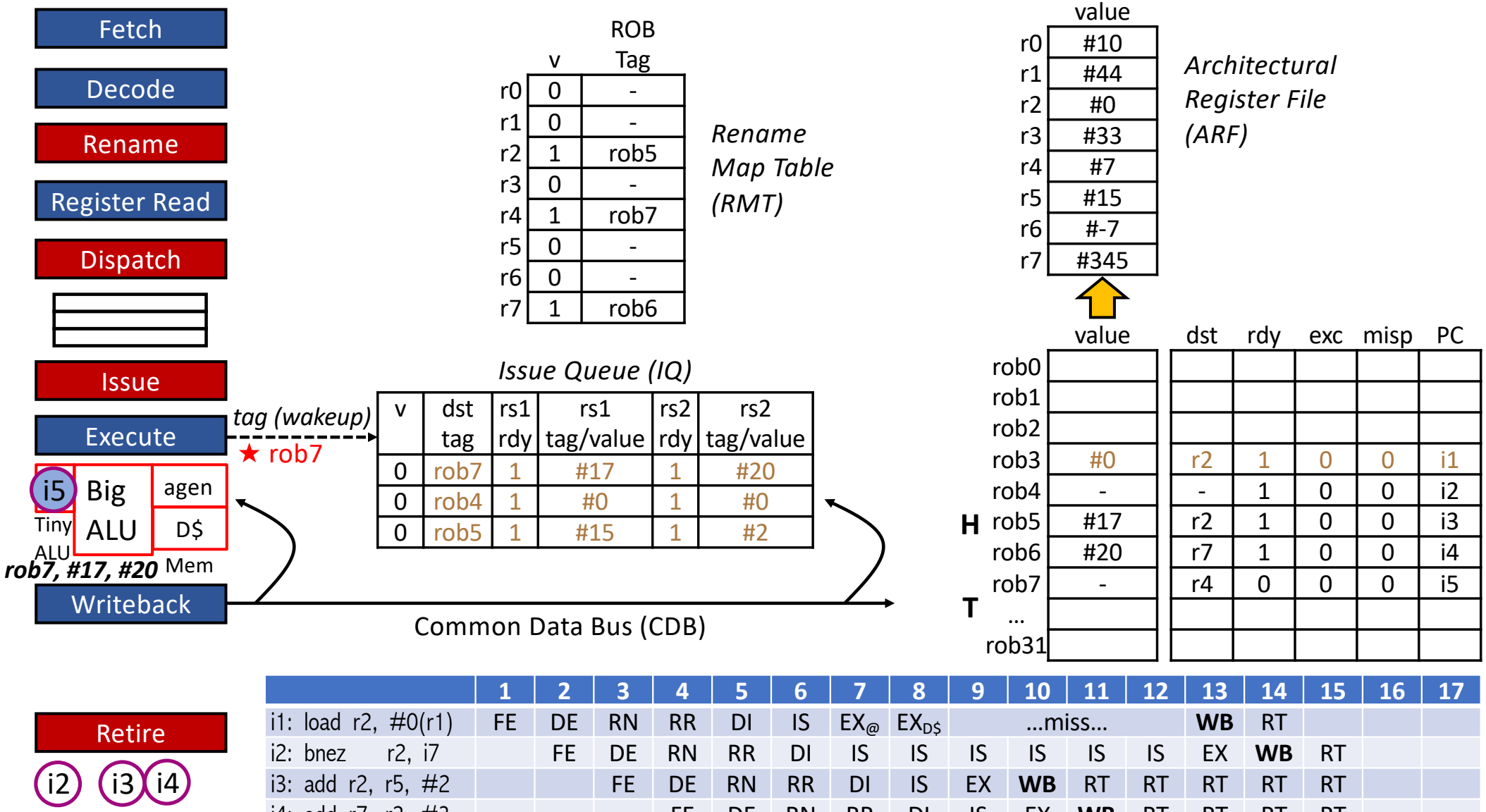
	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
i1: load r2, #0(r1)	FE	DE	RN	RR	DI	IS	$EX_{@}$	$EX_{D\$}$	...miss...								
i2: bnez r2, i7		FE	DE	RN	RR	DI	IS	IS	IS	IS	IS	IS					
i3: add r2, r5, #2			FE	DE	RN	RR	DI	IS	EX	**WB**	RT	RT					
i4: add r7, r2, #3				FE	DE	RN	RR	DI	IS	EX	**WB**	RT					



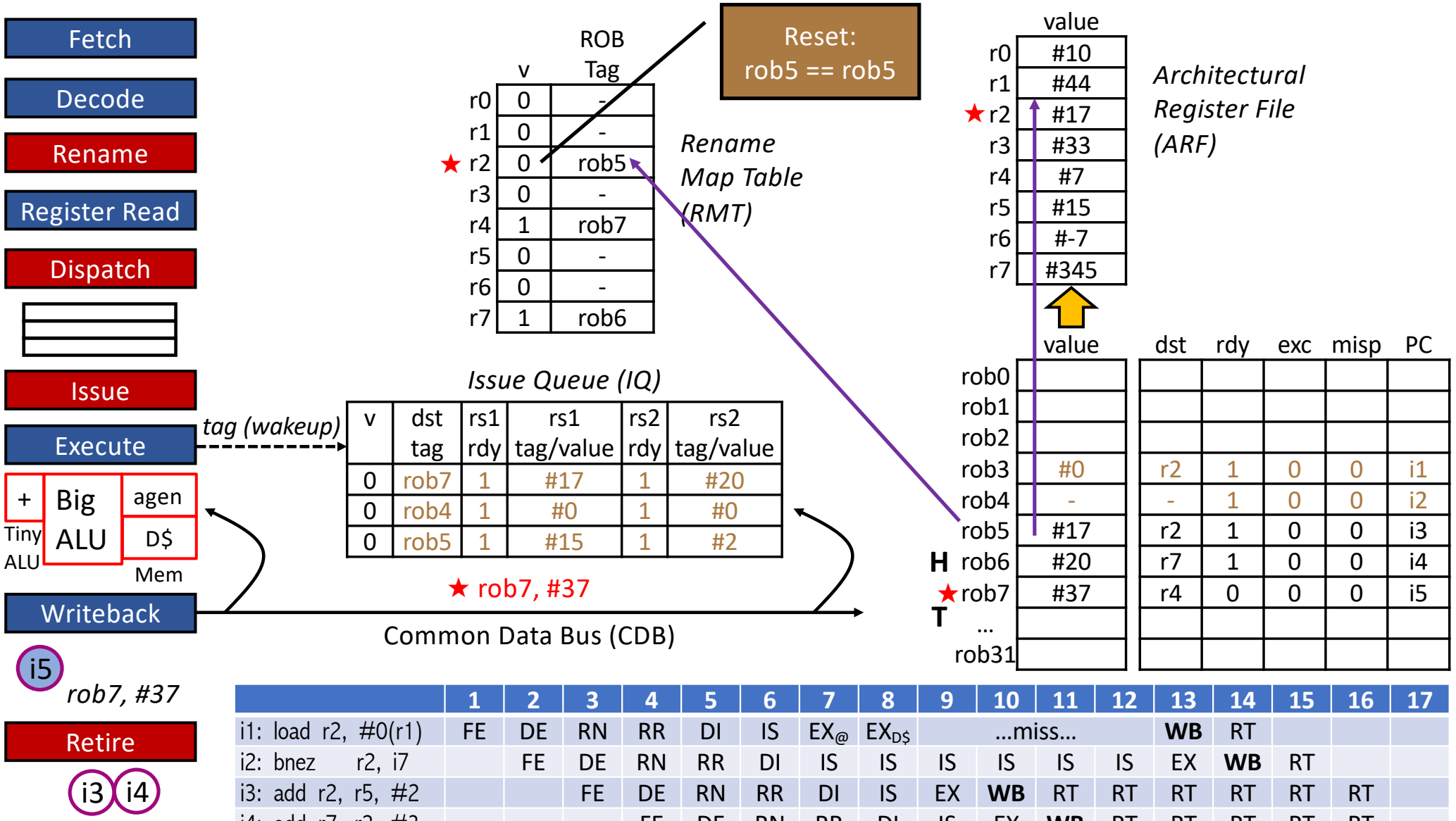
	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
i1: load r2, #0(r1)	FE	DE	RN	RR	DI	IS	$EX_{@}$	$EX_{D\$}$	...miss...				**WB**				
i2: bnez r2, i7		FE	DE	RN	RR	DI	IS	IS	IS	IS	IS	IS	EX				
i3: add r2, r5, #2			FE	DE	RN	RR	DI	IS	EX	**WB**	RT	RT	RT				
i4: add r7, r2, #3				FE	DE	RN	RR	DI	IS	EX	**WB**	RT	RT				



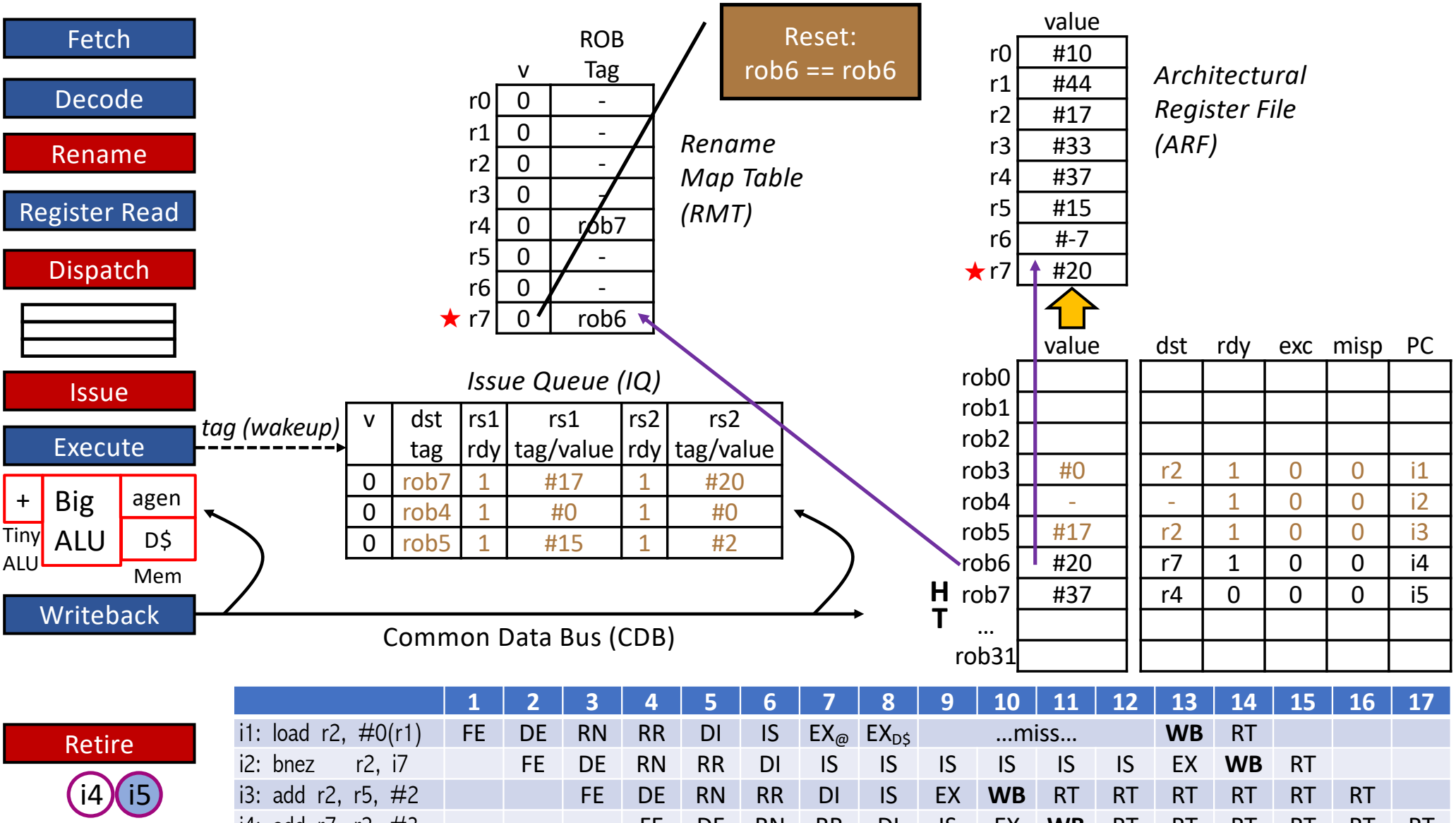
	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
i1: load r2, #0(r1)	FE	DE	RN	RR	DI	IS	$EX_{@}$	$EX_{D\$}$	...miss...				**WB**	RT			
i2: bnez r2, i7		FE	DE	RN	RR	DI	IS	IS	IS	IS	IS	IS	EX	**WB**			
i3: add r2, r5, #2			FE	DE	RN	RR	DI	IS	EX	**WB**	RT	RT	RT	RT			
i4: add r7, r2, #3				FE	DE	RN	RR	DI	IS	EX	**WB**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



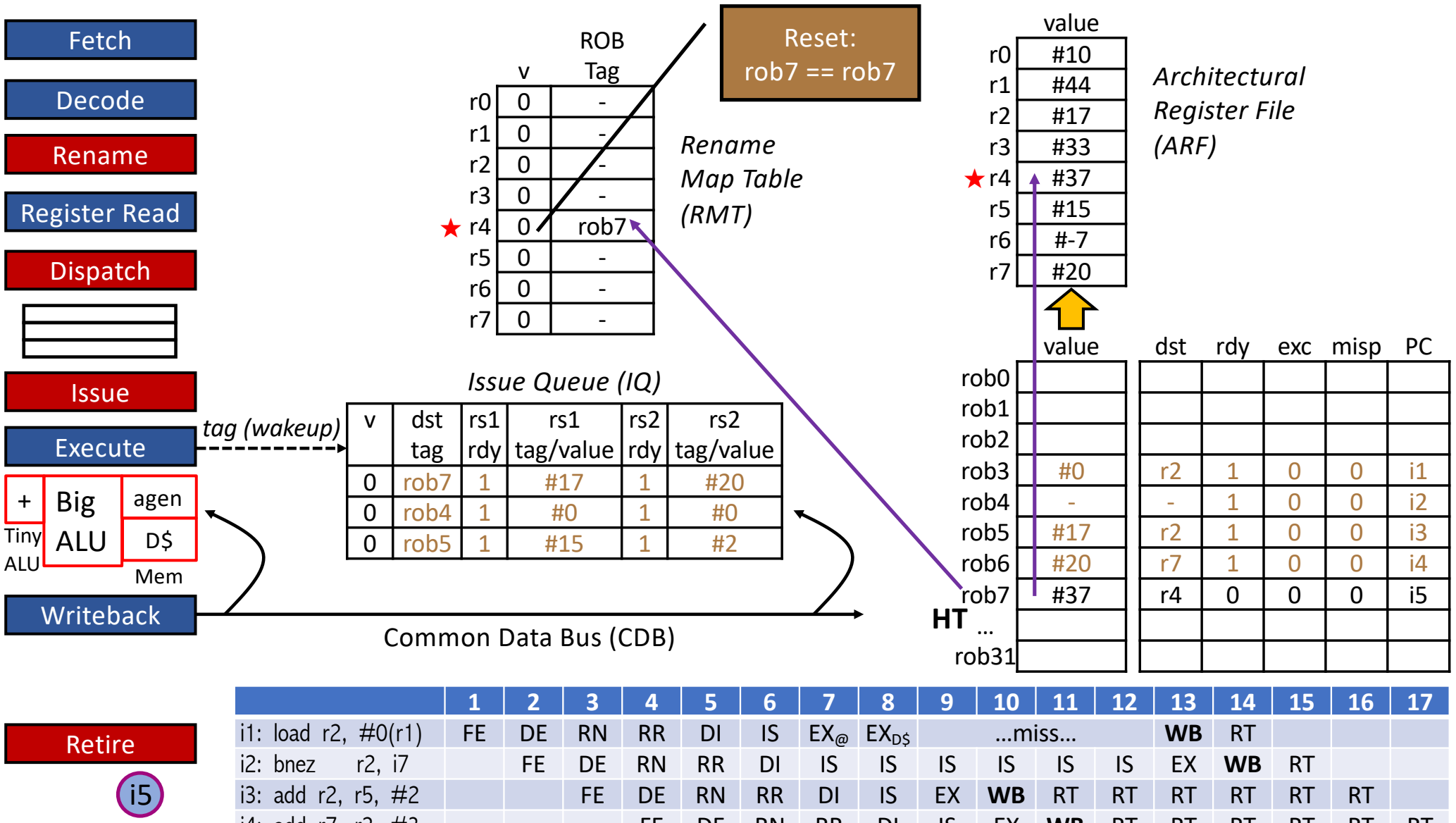
	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
i1: load r2, #0(r1)	FE	DE	RN	RR	DI	IS	$EX_{@}$	$EX_{D\$}$	...miss...				**WB**	RT			
i2: bnez r2, i7		FE	DE	RN	RR	DI	IS	IS	IS	IS	IS	IS	EX	**WB**	RT		
i3: add r2, r5, #2			FE	DE	RN	RR	DI	IS	EX	**WB**	RT	RT	RT	RT	RT		
i4: add r7, r2, #3				FE	DE	RN	RR	DI	IS	EX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



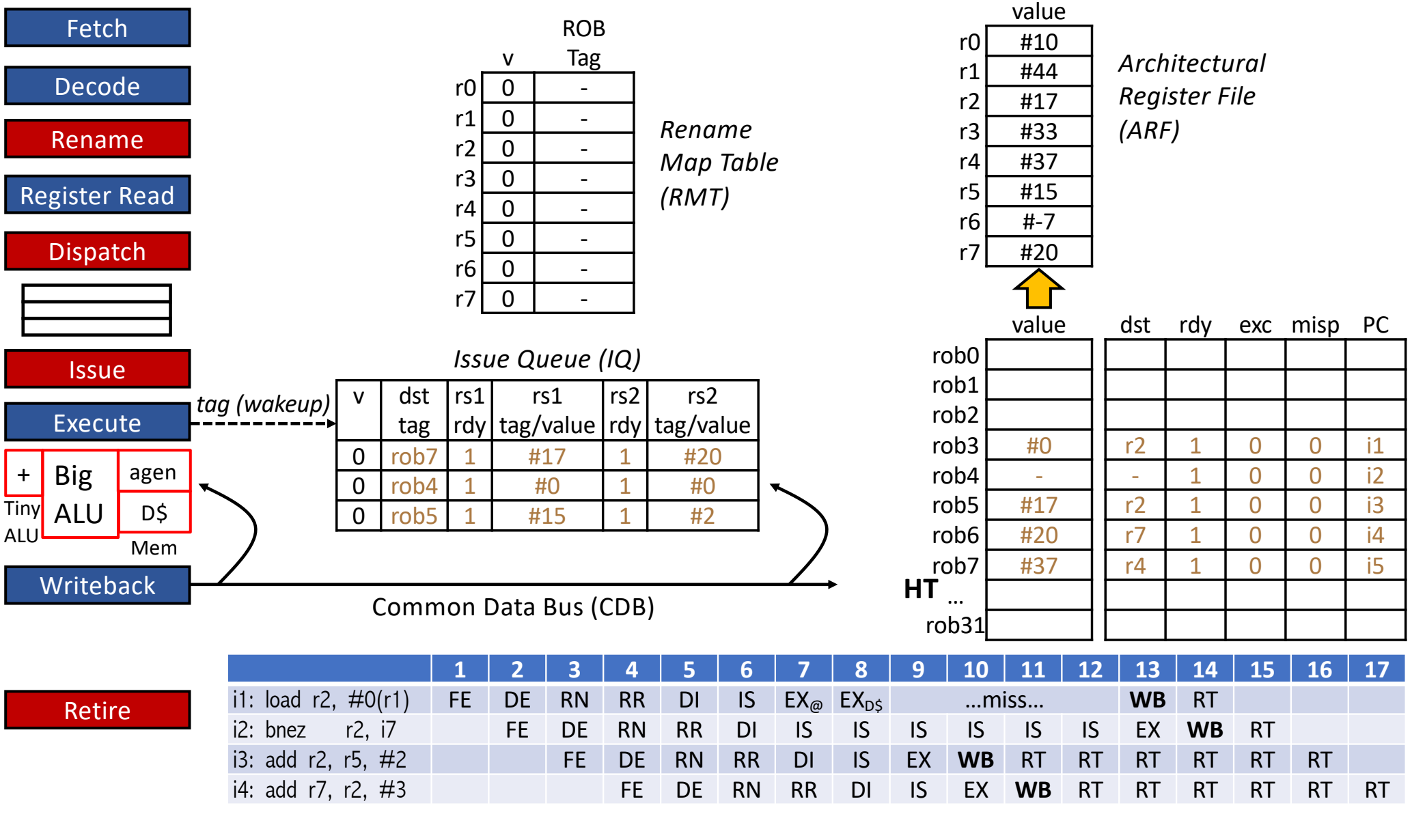
	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
i1: load r2, #0(r1)	FE	DE	RN	RR	DI	IS	$EX_{@}$	$EX_{D\$}$	...miss...				**WB**	RT			
i2: bnez r2, i7		FE	DE	RN	RR	DI	IS	IS	IS	IS	IS	IS	EX	**WB**	RT		
i3: add r2, r5, #2			FE	DE	RN	RR	DI	IS	EX	**WB**	RT	RT	RT	RT	RT	RT	
i4: add r7, r2, #3				FE	DE	RN	RR	DI	IS	EX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



	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
i1: load r2, #0(r1)	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



	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
i1: load r2, #0(r1)	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



	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
i1: load r2, #0(r1)	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

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

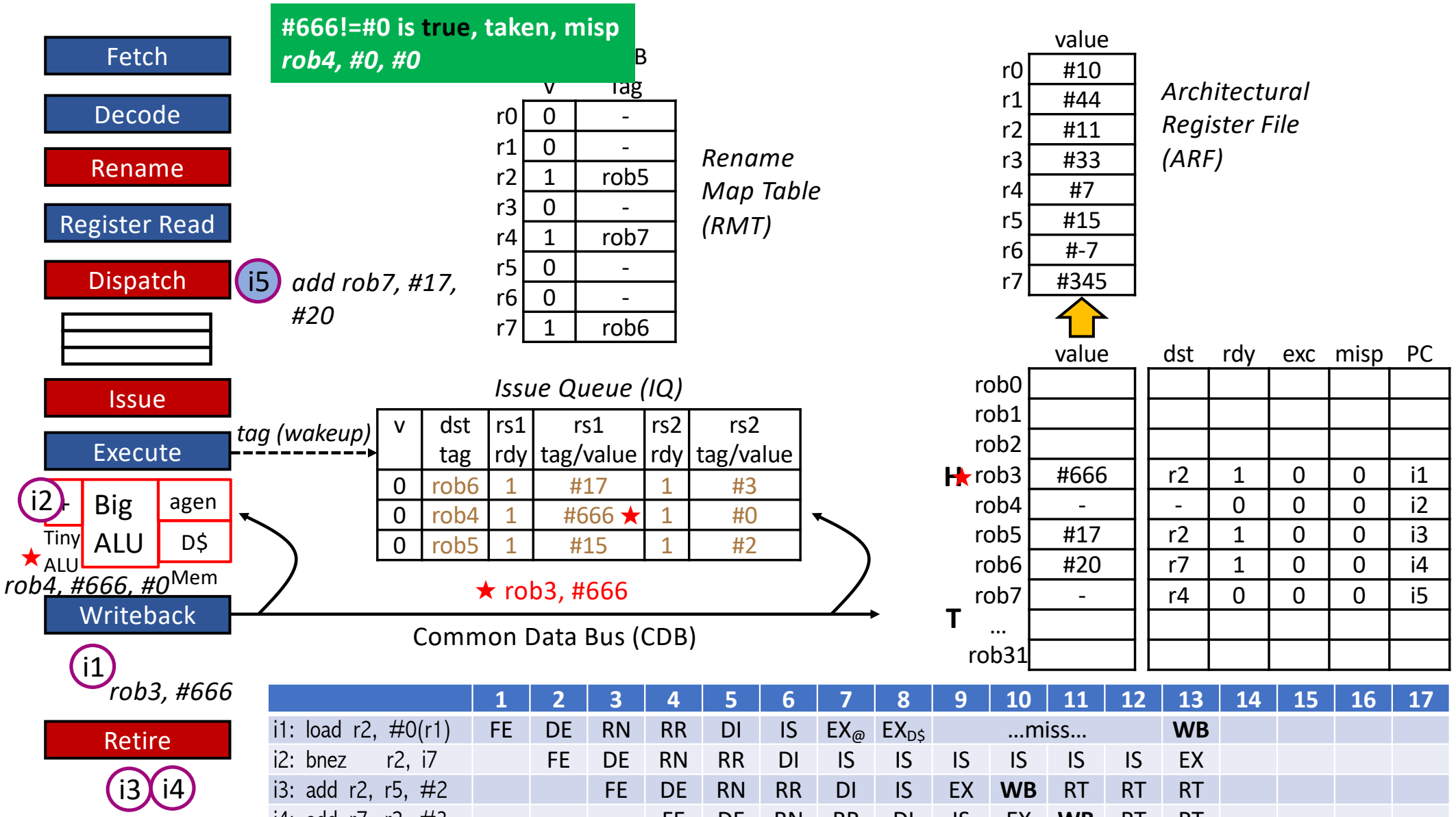
- **Compared to scoreboard only**
  - ROB did not degrade performance
  - Fetch did not stall as before (tolerated D$ miss)
  - In-order retirement did not impede OOO, speculative execution

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

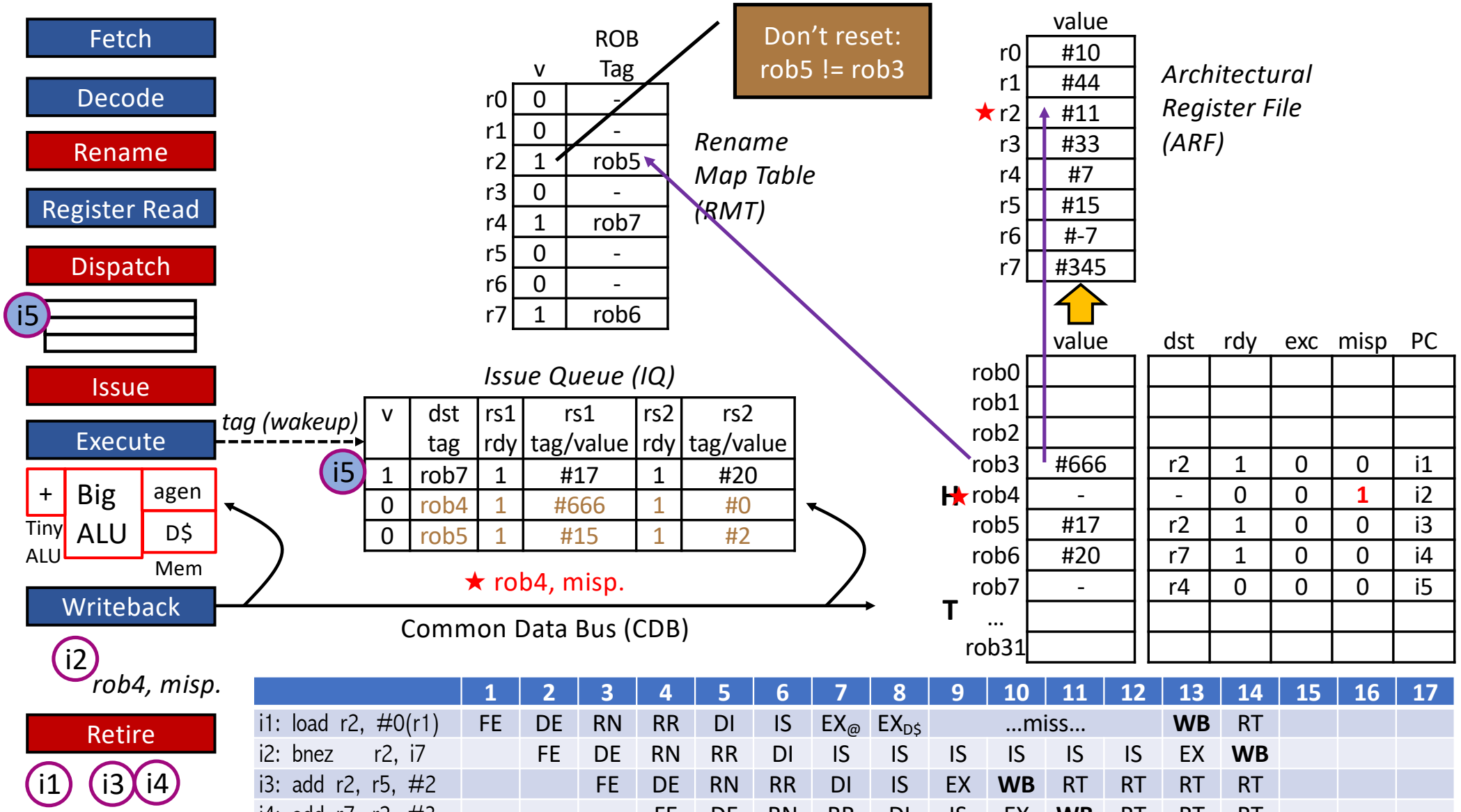
Recovery

Revise the previous scenario assuming mispredicted branch

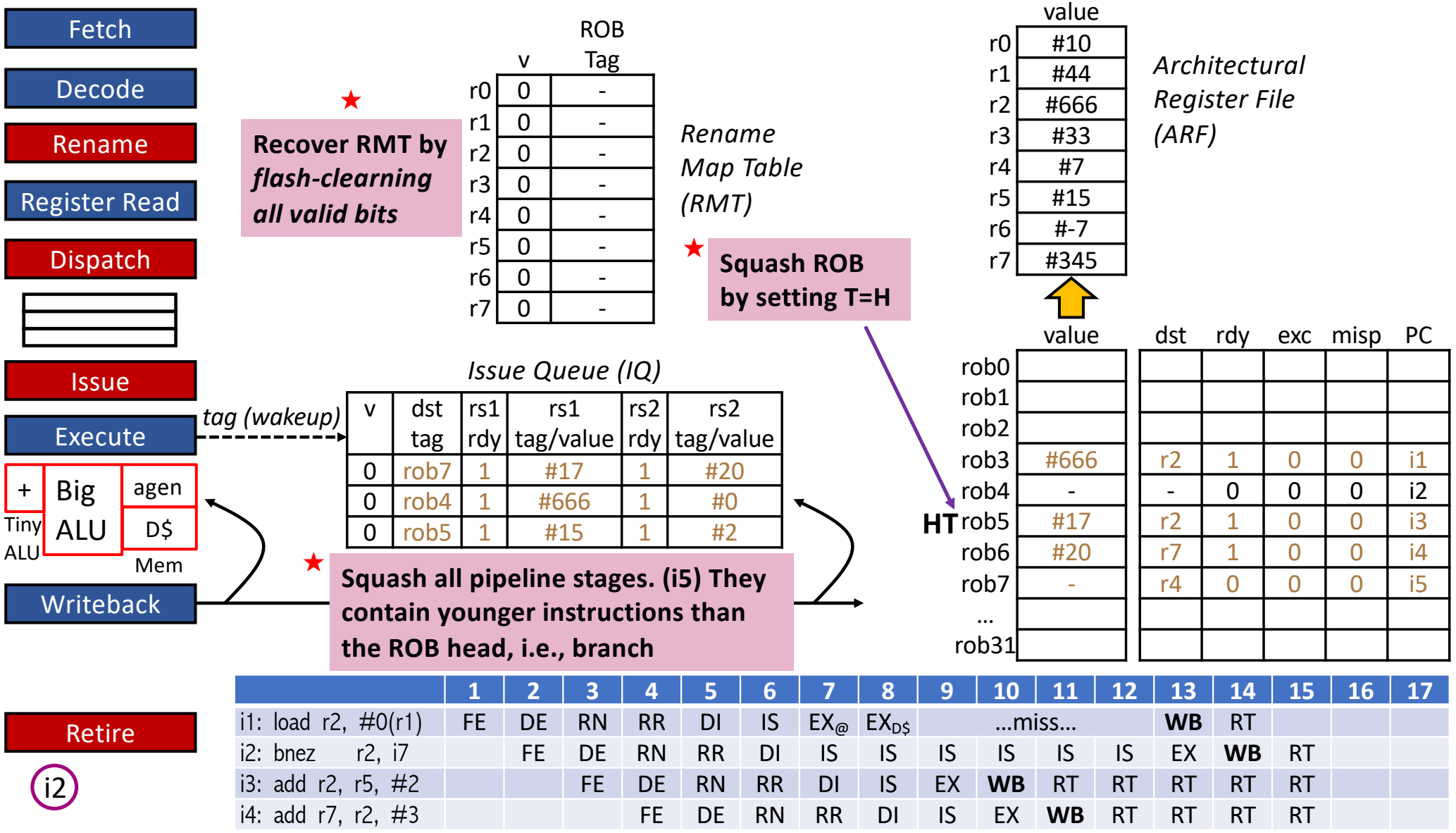
- i1 (load instruction) gets the value #666 instead of #0



	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
i1: load r2, #0(r1)	FE	DE	RN	RR	DI	IS	$EX_{@}$	$EX_{D\$}$	...miss...				**WB**				
i2: bnez 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				

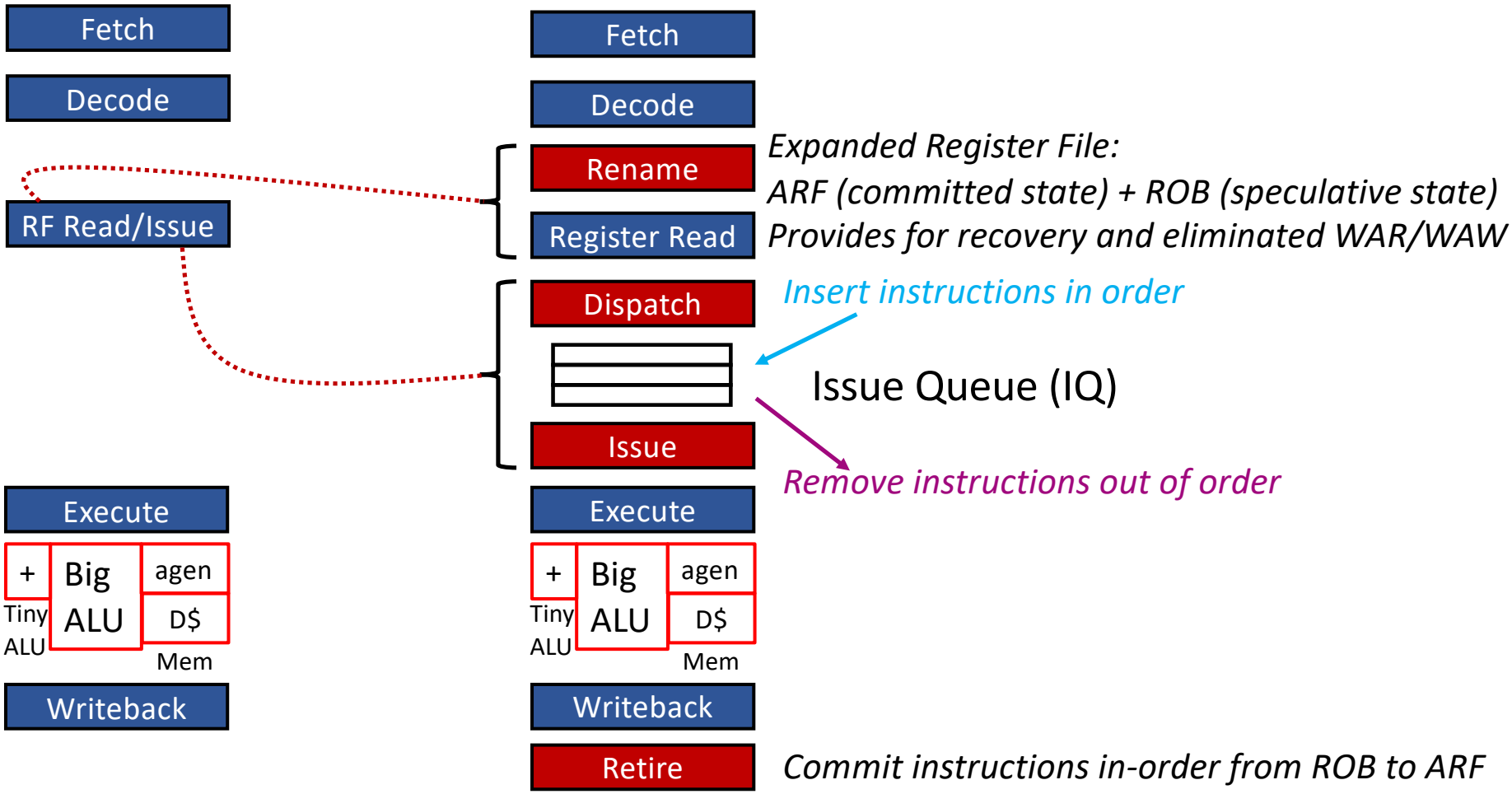


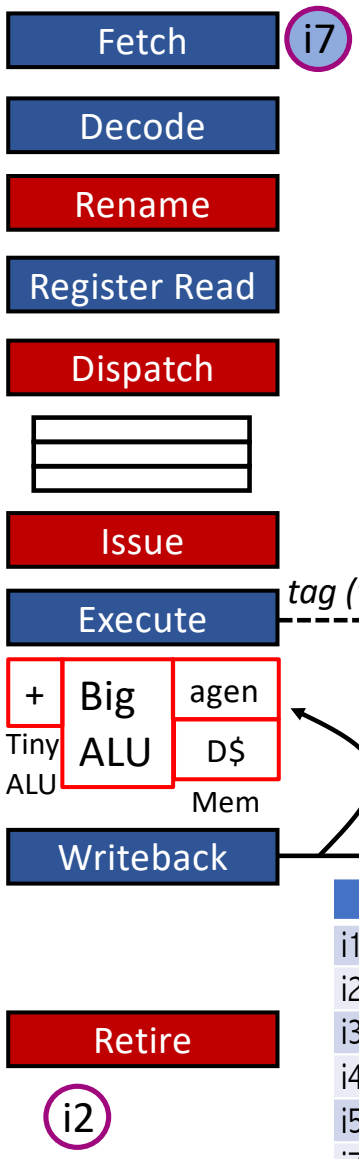
	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
i1: load r2, #0(r1)	FE	DE	RN	RR	DI	IS	$EX_{@}$	$EX_{D\$}$	...miss...				**WB**	RT			
i2: bnez r2, i7		FE	DE	RN	RR	DI	IS	IS	IS	IS	IS	IS	EX	**WB**			
i3: add r2, r5, #2			FE	DE	RN	RR	DI	IS	EX	**WB**	RT	RT	RT	RT			
i4: add r7, r2, #3				FE	DE	RN	RR	DI	IS	EX	**WB**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



	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
i1: load r2, #0(r1)	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		
i4: add r7, r2, #3				FE	DE	RN	RR	DI	IS	EX	**WB**	RT	RT	RT	RT		

In-order to Out-of-Order





ROB

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

Rename Map Table (RMT)

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

Architectural Register File (ARF)

Issue Queue (IQ)

tag (wakeup)

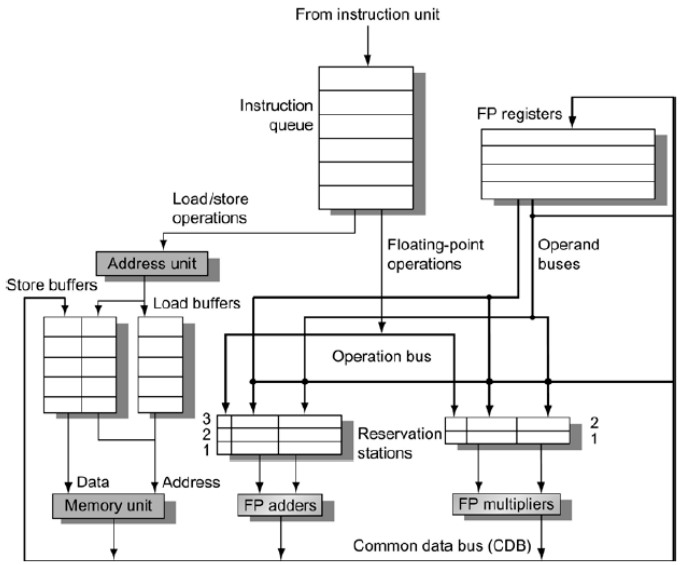
v	dst tag	rs1 rdy	rs1 tag/value	rs2 rdy	rs2 tag/value
0	rob7	1	#17	1	#20
0	rob4	1	#666	1	#0
0	rob5	1	#15	1	#2

	value	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: load r2, #0(r1)	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		
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	

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)



Renaming with Reservation Stations in 360/91

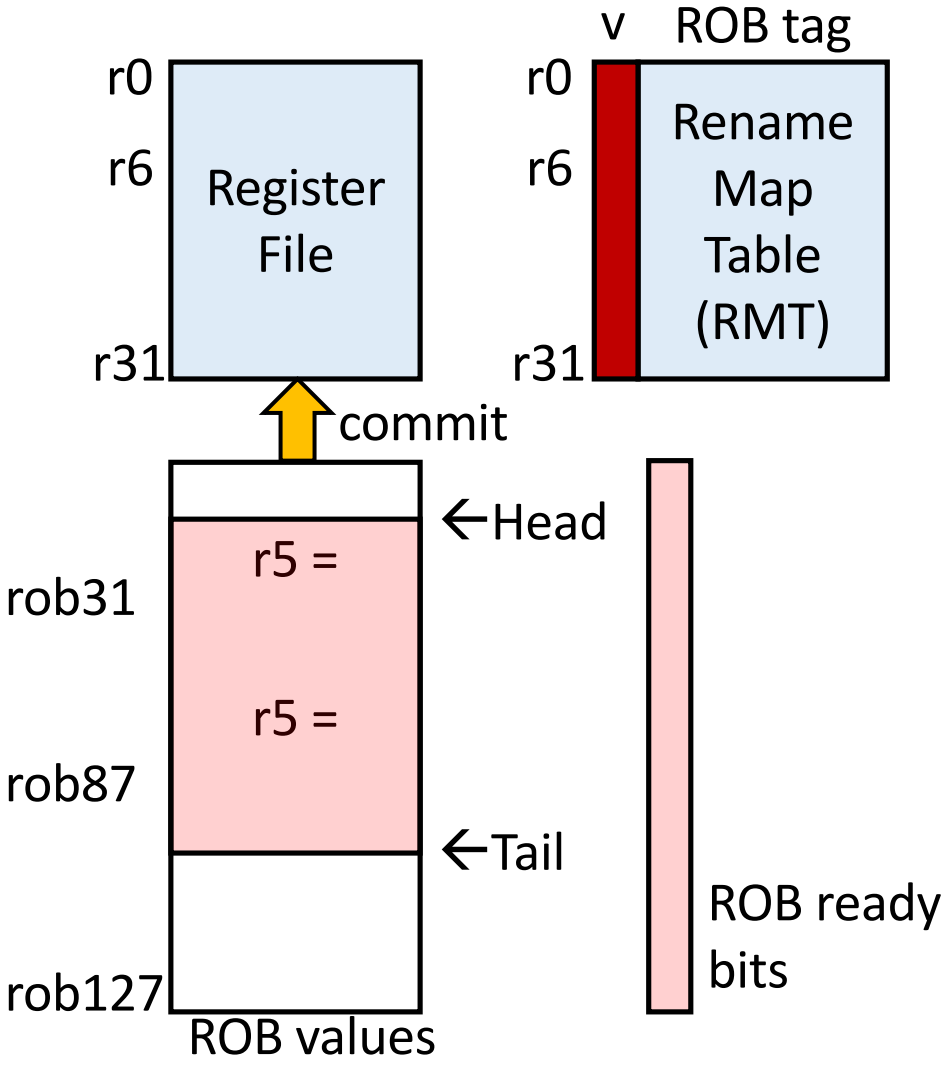
- Tomasulo's Algorithm [1967]
  - Reservation stations (RS) are used to extend the register file
  - Each RS entry has a unique tag
  - Results are forwarded over the CDB
  - Section 3.4: HP, A Quantitative Approach

		Instruction status		
Instruction		Issue	Execute	Write result
fld	f6,32(x2)	√	√	√
fld	f2,44(x3)	√	√	
fmul.d	f0,f2,f4	√		
fsub.d	f8,f2,f6	√		
fdiv.d	f0,f0,f6	√		
fadd.d	f6,f8,f2	√		

	Reservation stations						
Name	Busy	Op	Vj	Vk	Qj	Qk	A
Load1	No						
Load2	Yes	Load					44 + Regs[x3]
Add1	Yes	SUB		Mem[32 + Regs[x2]]	Load2		
Add2	Yes	ADD			Add1	Load2	
Add3	No						
Mult1	Yes	MUL		Regs[f4]	Load2		
Mult2	Yes	DIV		Mem[32 + Regs[x2]]	Mult1		

	Register status								
Field	f0	f2	f4	f6	f8	f10	f12	...	f30
Qi	Mult1	Load2		Add2	Add1	Mult2			

ARF + ROB Summary



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

Recovery

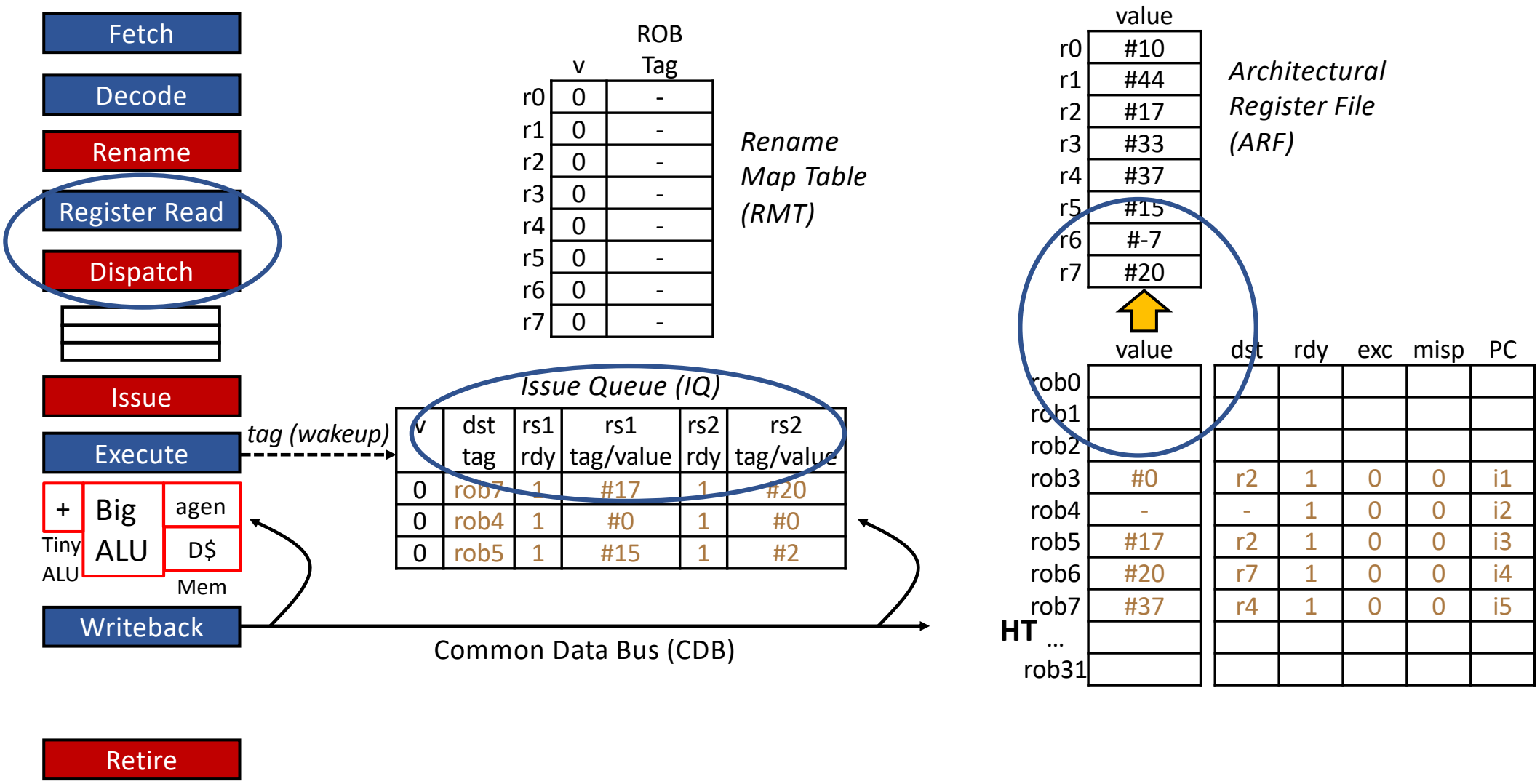
- Wait until exception/misprediction reaches head
- T = H
- Reset all “v” bits in RMT

Revision: Main Concepts

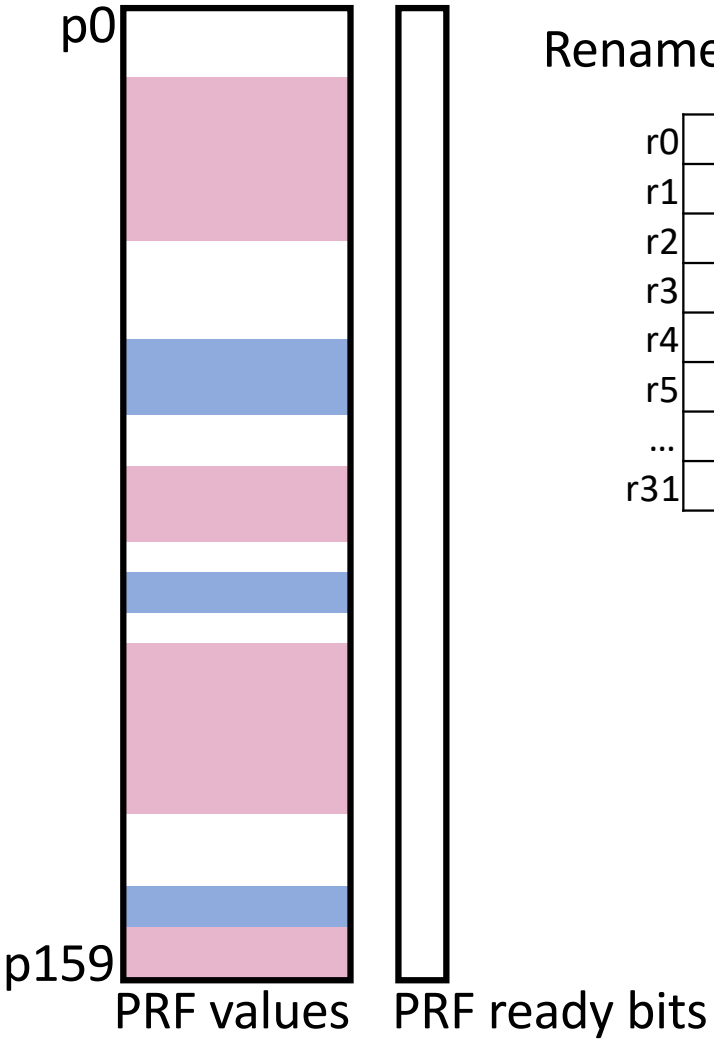
- Register renaming
  - Rename logical registers to an extended set of physical registers
  - Avoid WAR and WAW hazards (main structure: ROB or RS/IQ)
- Dynamic scheduling
  - Send instructions to the functional units out of the original program order (IQ)
- Speculation
  - Predict branch outcomes and 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


Fetch
Decode
Rename
Register Read
Dispatch
Issue
Execute
+
Big
ALU
agen
D$
Tiny
ALU
Mem
Writeback
Retire
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 #0 1 #0
0 rob5 1 #15 1 #2
Common Data Bus (CDB)
value
r0 #10
r1 #44
r2 #17
r3 #33
r4 #37
r5 #15
r6 #-7
r7 #20
Architectural
Register File
(ARF)
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

PRF Style



Rename Map Table

r0	p10
r1	p67
r2	p11
r3	p33
r4	p46
r5	
…	
r31	p2

Phys. Reg. Tag

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.

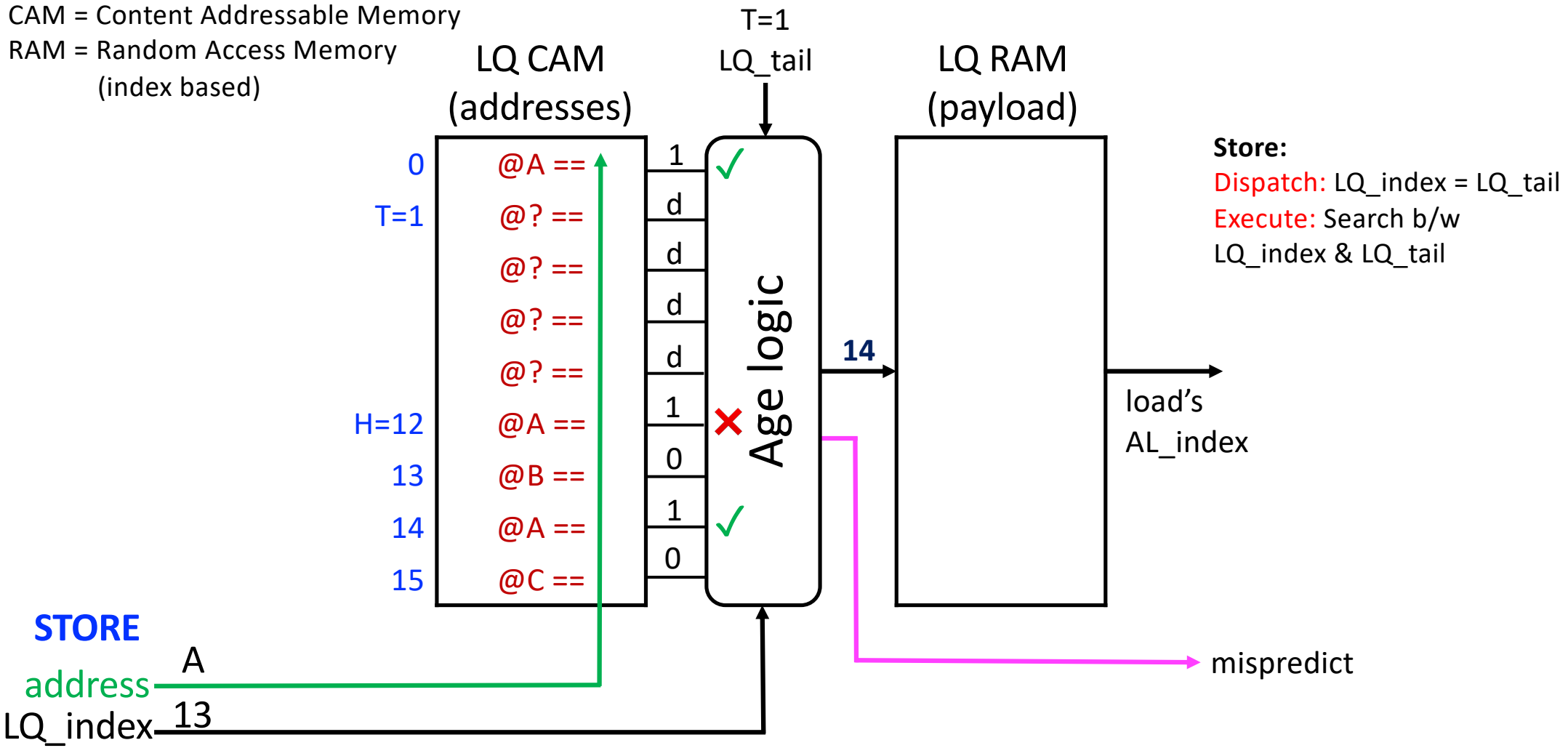
The die area savings are key as they enable one of Sandy Bridge's major innovations: AVX performance.

Loads and Stores

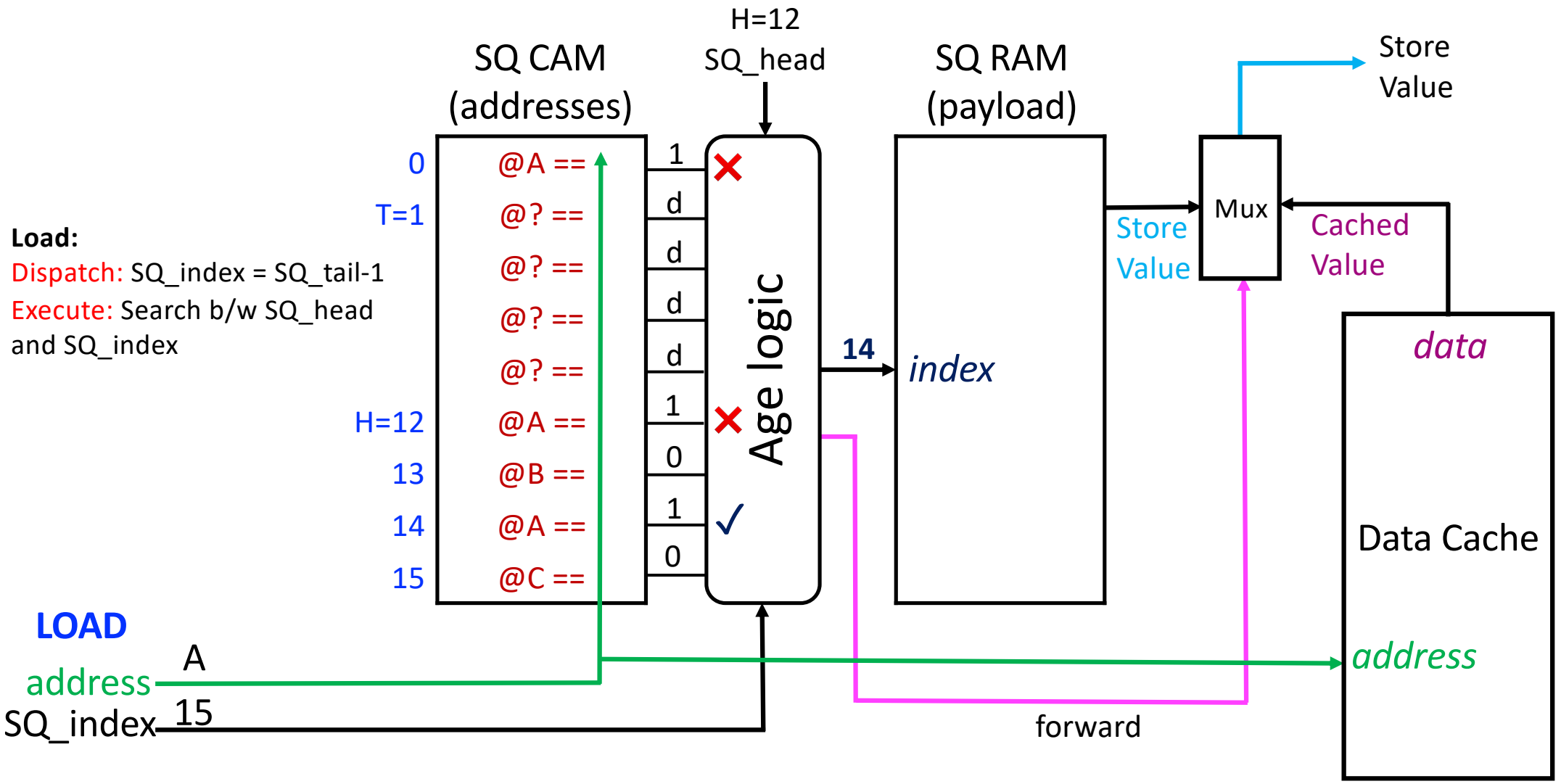
- Loads and stores also execute out of order
- Store cancels all speculative (younger) loads with matching addresses
- Load searches for all speculative (older) stores with matching addresses
  - It gets the best it can (cache, main memory, ROB, RF)
- Once we have speculation support, we can predict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

Store Execution Datapath

CAM = Content Addressable Memory
RAM = Random Access Memory
(index based)



Load Execution Datapath



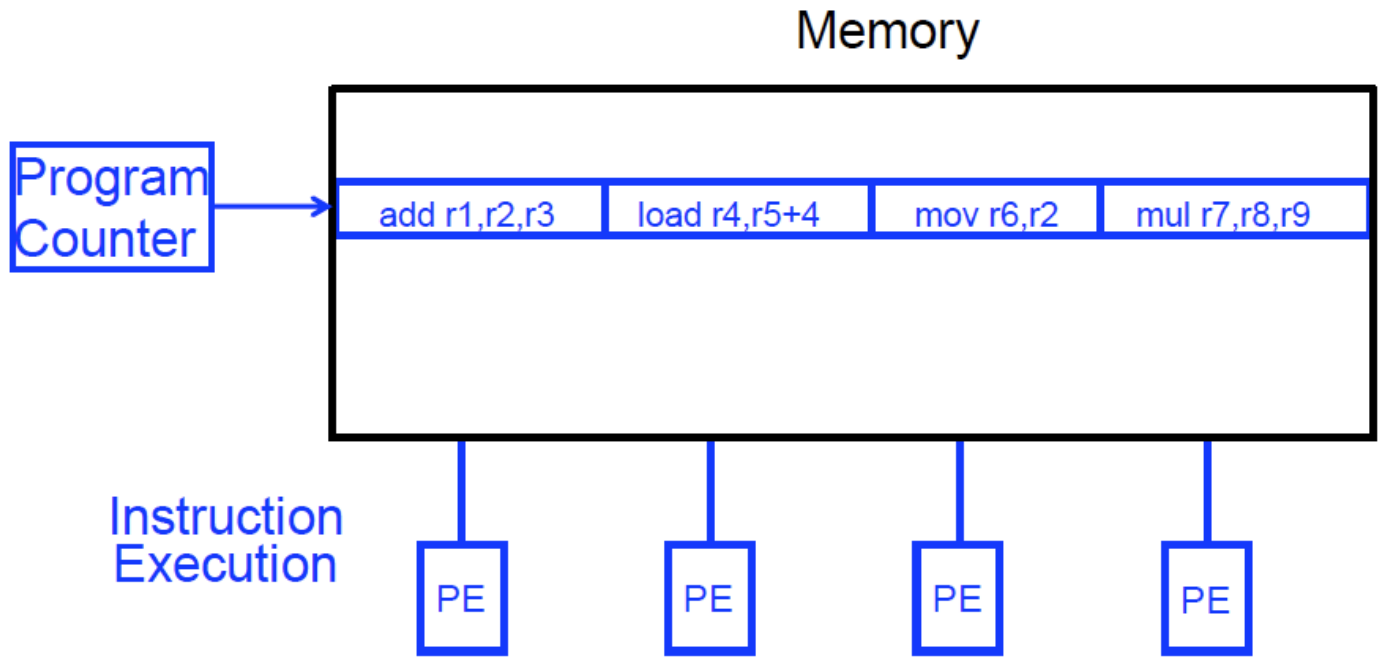
Compilation Techniques for Exploiting ILP

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, which we will see soon)

- 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

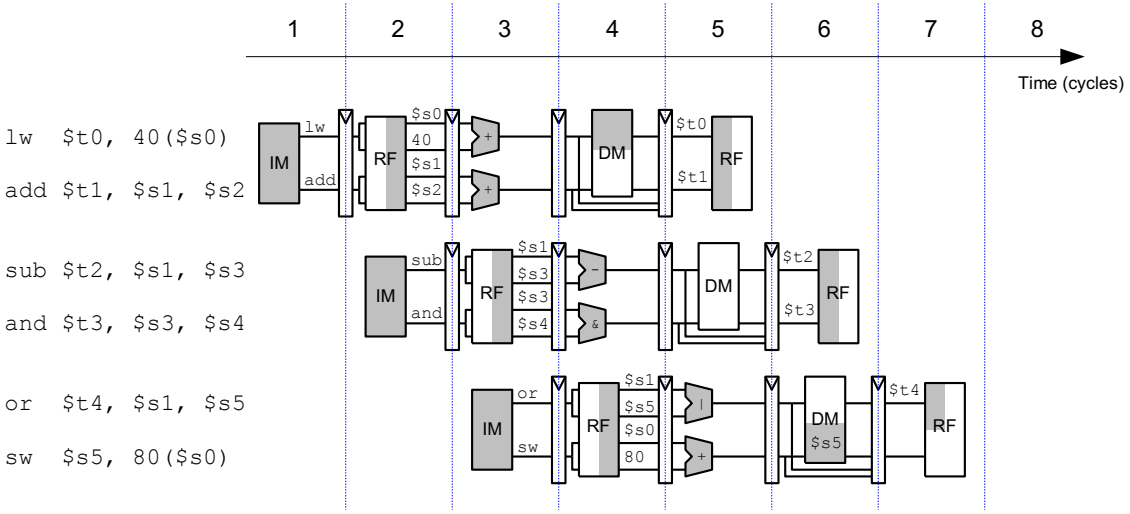
  3. Instructions in a bundle statically aligned to be directly supplied into the functional units

VLIW Performance Example (2-wide bundles)

```
lw  $t0, 40($s0)     add $t1, $s1, $s2
sub $t2, $s1, $s3    and $t3, $s3, $s4
or  $t4, $s1, $s5    sw  $s5, 80($s0)
```

Bundle 1
Bundle 2
Bundle 3

Ideal IPC = 2



Actual IPC = 2 (6 instructions issued in 3 cycles)

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

Proceedings of the ACM SIGPLAN '84 Symposium on Compiler Construction,
SIGPLAN Notices Vol. 19, No. 6, June 1984

Parallel Processing: A Smart Compiler and a Dumb Machine

**Joseph A. Fisher, John R. Ellis,
John C. Ruttenberg, and Alexandru Nicolau**

**Department of Computer Science, Yale University
New Haven, CT 06520**

Abstract

Multiprocessors and vector machines, the only successful parallel architectures, have coarse-grained parallelism that is hard for compilers to take advantage of. We've developed a new fine-grained parallel architecture and a compiler that together offer order-of-magnitude speedups for ordinary scientific code.

future, and we're building a VLIW machine, the ELI (Enormously Long Instructions) to prove it.

In this paper we'll describe some of the compilation techniques used by the Bulldog compiler. The ELI project and the details of Bulldog are described elsewhere [4, 6, 7, 15, 17].

Fisher et al., "Parallel Processing: A Smart Compiler and a Dumb Machine," CC 1984.

VLIW Philosophy & Principles

- 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

VLIW Philosophy and Properties

More formally, VLIW architectures have the following properties:

There is one central control unit issuing a single long instruction per cycle.

Each long instruction consists of many tightly coupled independent operations.

Each operation requires a small, statically predictable number of cycles to execute.

Operations can be pipelined. These properties distinguish VLIWs from multiprocessors (with large asynchronous tasks) and dataflow machines (without a single flow of control, and without the tight coupling). VLIWs have none of the required regularity of a vector processor, or true array processor.

Fisher, "Very Long Instruction Word architectures and the ELI-512," ISCA 1983.

Commercial VLIW Machines

- Multiflow TRACE, Josh Fisher (7-wide, 28-wide)
- Cydrome Cydra 5, Bob Rau
- Transmeta Crusoe: x86 binary-translated into internal VLIW
- TI C6000, Trimedia, STMicro (DSP & embedded processors) and some ATI/AMD GPUs
  - Most successful commercially

- Intel IA-64
  - Not fully VLIW, but based on VLIW principles
  - EPIC (Explicitly Parallel Instruction Computing)
  - Instruction bundles can have dependent instructions
  - A few bits in the instruction format specify explicitly which instructions in the bundle are dependent on which other ones

VLIW Tradeoffs

- Advantages
  - + No need for dynamic scheduling hardware → simple hardware
  - + No need for dependency checking within a VLIW instruction → simple hardware for multiple instruction issue + no renaming
  - + No need for instruction alignment/distribution after fetch to different functional units → simple hardware

- Disadvantages
  - -- Compiler needs to find N independent operations per cycle
    - -- If it cannot, inserts NOPs in a VLIW instruction
    - -- Parallelism loss AND code size increase
  - -- Recompilation required when execution width (N), instruction latencies, functional units change (Unlike superscalar processing)
  - -- Lockstep execution causes independent operations to stall
    - -- No instruction can progress until the longest-latency instruction completes

VLIW Summary

- VLIW simplifies hardware, but requires complex compiler techniques
- Solely-compiler approach of VLIW has several downsides that reduce performance

  -- No tolerance for variable or long-latency operations (lock step)

  -- Too many NOPs (not enough parallelism discovered)

  -- Static schedule intimately tied to microarchitecture

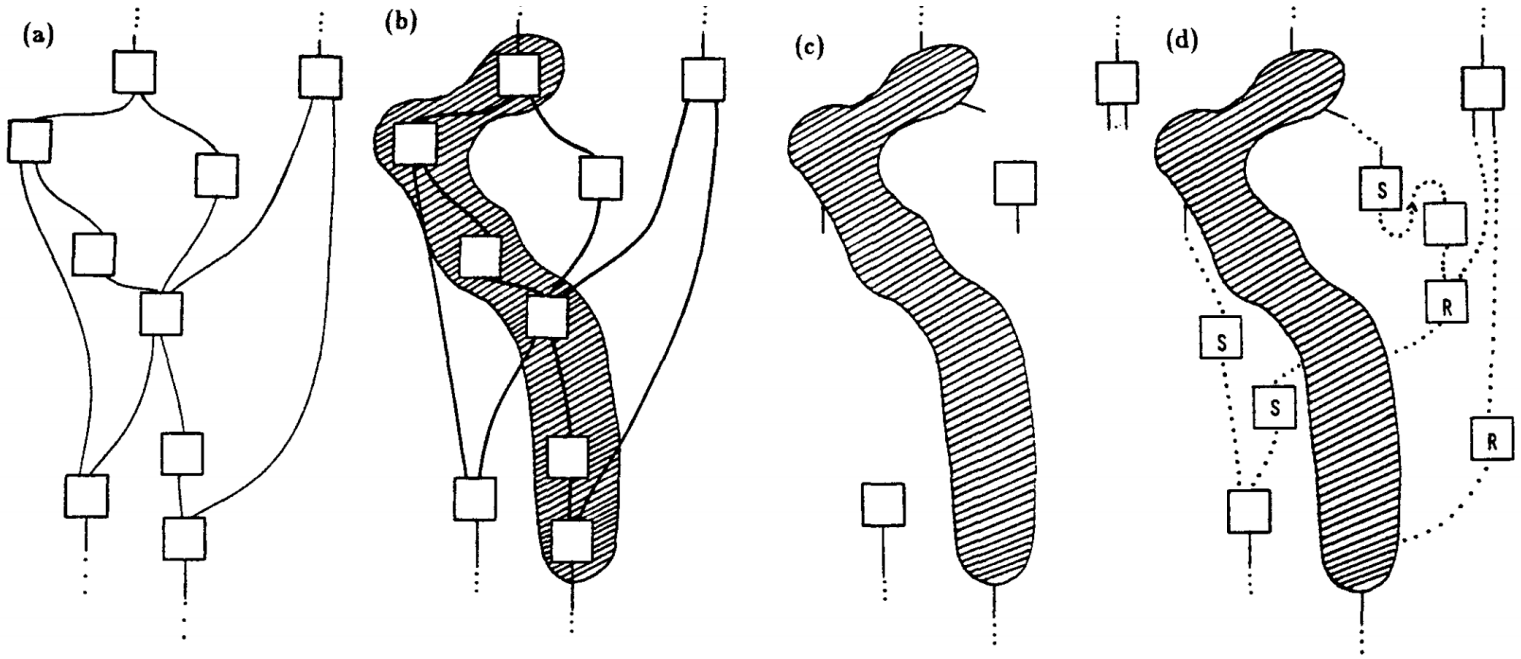
    -- Code optimized for one generation performs poorly for next

++ Most compiler optimizations developed for VLIW employed in optimizing compilers (for superscalar compilation)

- Enable code optimizations

++ VLIW very successful when parallelism is easier to find by the compiler (traditionally embedded markets, DSPs, GPUs)

Example Work: Trace Scheduling



TRACE SCHEDULING LOOP-FREE CODE

(a) A flow graph, with each block representing a basic block of code. (b) A trace picked from the flow graph. (c) The trace has been scheduled but it hasn't been relinked to the rest of the code. (d) The sections of unscheduled code that allow relinking.

Fisher, "Very Long Instruction Word architectures and the ELI-512," ISCA 1983.

Recommended Paper

VERY LONG INSTRUCTION WORD
ARCHITECTURES
AND THE ELI-512

JOSEPH A. FISHER
YALE UNIVERSITY
NEW HAVEN, CONNECTICUT 06520

ABSTRACT

By compiling ordinary scientific applications programs with a radical technique called trace scheduling, we are generating code for a parallel machine that will run these programs faster than an equivalent sequential machine — we expect 10 to 30 times faster.

Trace scheduling generates code for machines called Very Long Instruction Word architectures. In Very Long Instruction Word machines, many statically scheduled, tightly coupled, fine-grained operations execute in parallel within a single instruction stream. VLIWs are more parallel extensions of several current architectures.

These current architectures have never cracked a fundamental barrier. The speedup they get from parallelism is never more than a factor of 2 to 3. Not that we couldn't build more parallel machines of this type; but until trace scheduling we didn't know how to generate code for them. Trace scheduling finds sufficient parallelism in ordinary code to justify thinking about a highly parallel VLIW.

At Yale we are actually building one. Our machine, the ELI-512, has a horizontal instruction word of over 500 bits and

are presented in this paper. How do we put enough tests in each cycle without making the machine too big? How do we put enough memory references in each cycle without making the machine too slow?

WHAT IS A VLIW?

Everyone wants to use cheap hardware in parallel to speed up computation. One obvious approach would be to take your favorite Reduced Instruction Set Computer, let it be capable of executing 10 to 30 RISC-level operations per cycle controlled by a very long instruction word. (In fact, call it a VLIW.) A VLIW looks like very parallel horizontal microcode.

More formally, VLIW architectures have the following properties:

There is one central control unit issuing a single long instruction per cycle.

Each long instruction consists of many tightly coupled independent operations.

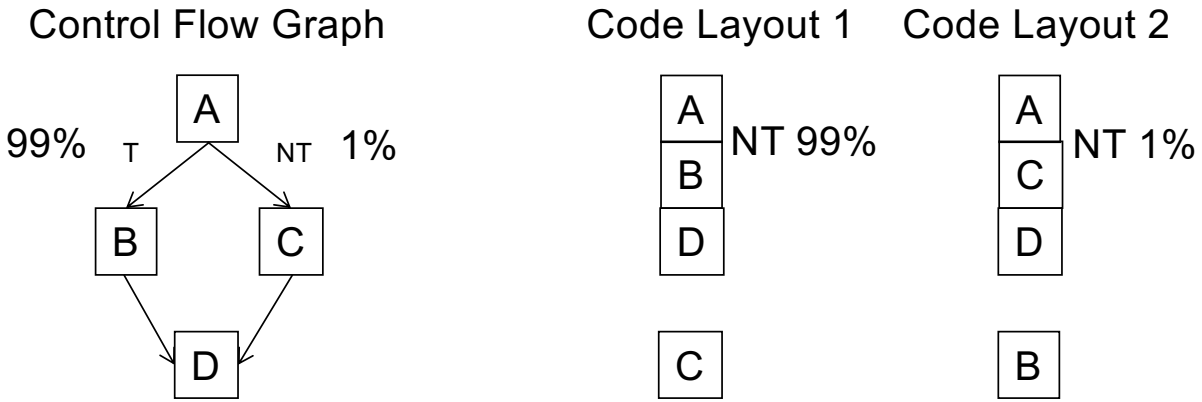
Each operation requires a small, statically predictable number of cycles to execute.

Operations can be pipelined. These properties distinguish

Fisher, "Very Long Instruction Word architectures and the ELI-512," ISCA 1983.

Recall: Basic Block Reordering

- Likely-taken branch instructions are a problem
  - They hurt the accuracy of "always not taken" branch prediction
  - They make static code reordering/scheduling difficult
- Idea: Convert likely-taken branch to a likely not-taken one
  - i.e., reorder basic blocks (after profiling)
  - Basic block: code with a single entry and single exit point



- Code Layout 1 leads to the fewest branch mispredictions

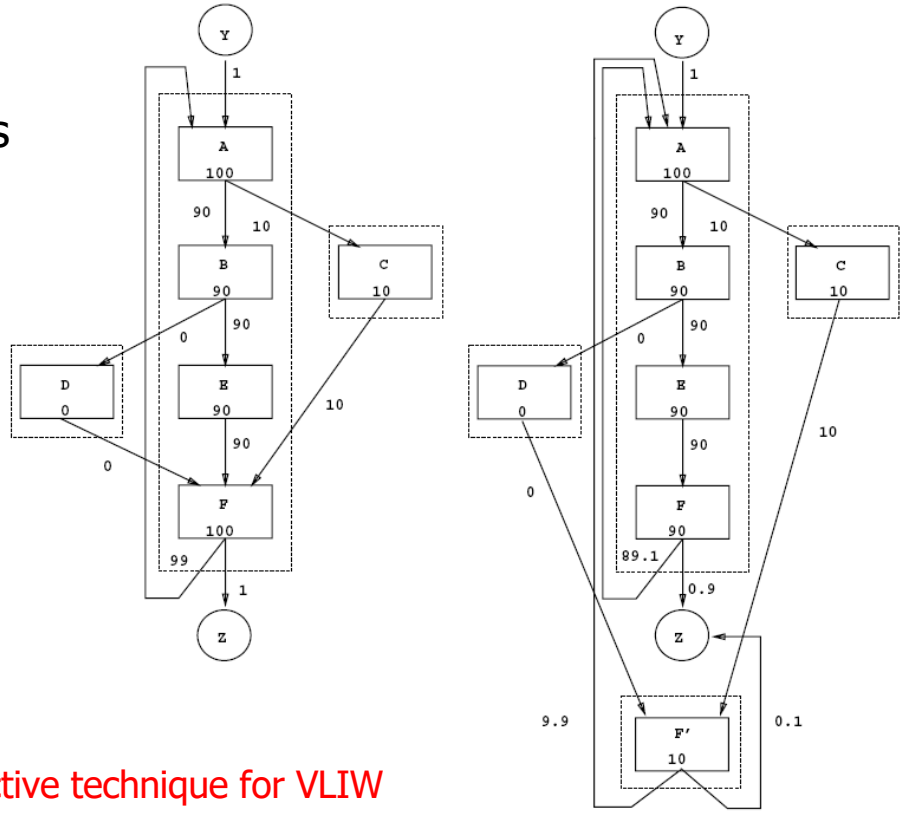
Pettis and Hansen, "Profile Guided Code Positioning," PLDI 1990.

Superblock: Can We Do Better?

- Idea: Combine frequently-executed basic blocks such that they form a single-entry multiple exit larger block, which is likely executed as straight-line code

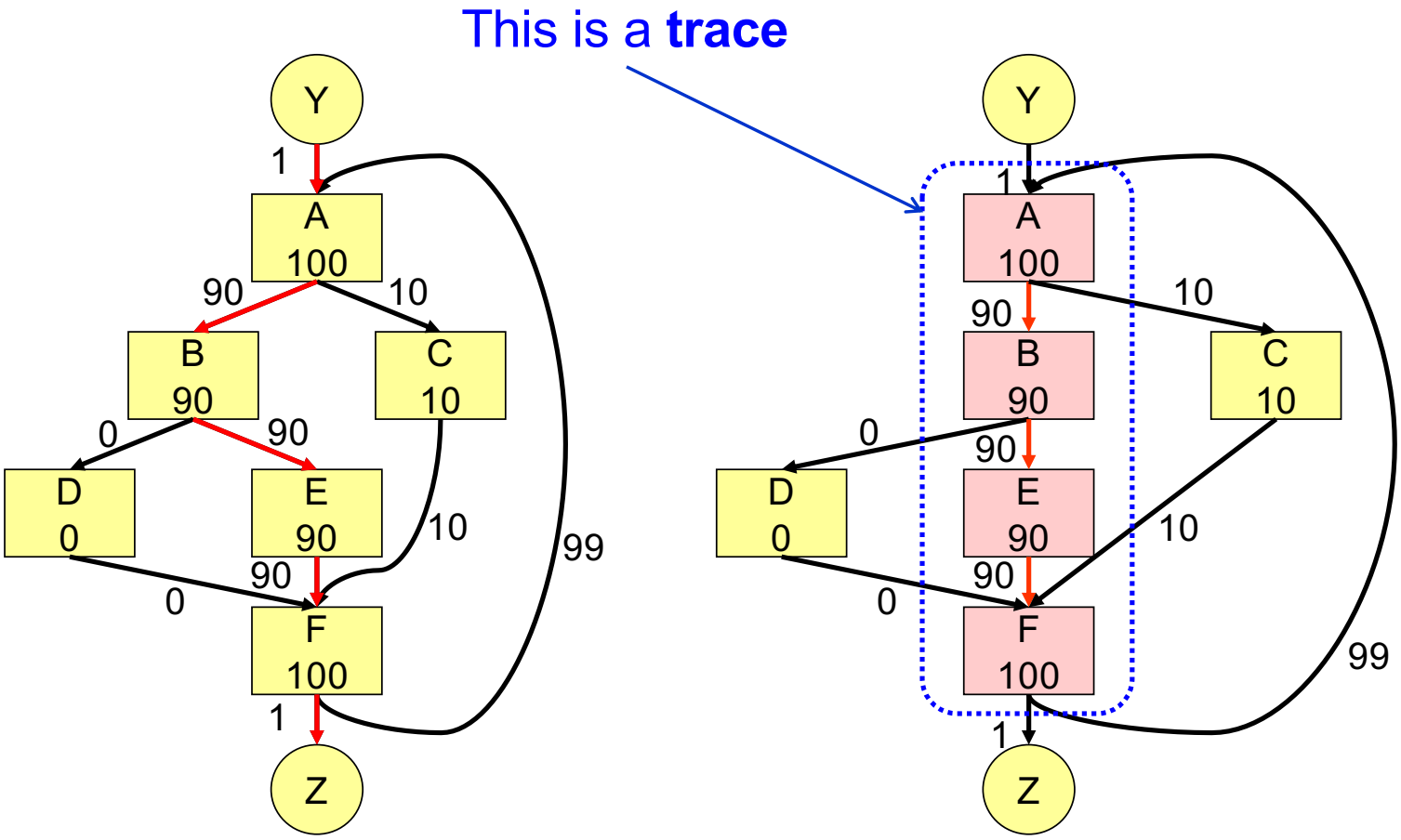
+ Reduces branch mispredictions
+ Enables aggressive compiler optimizations and code reordering within the superblock

-- Increased code size
-- Requires recompilation
-- Profile dependent

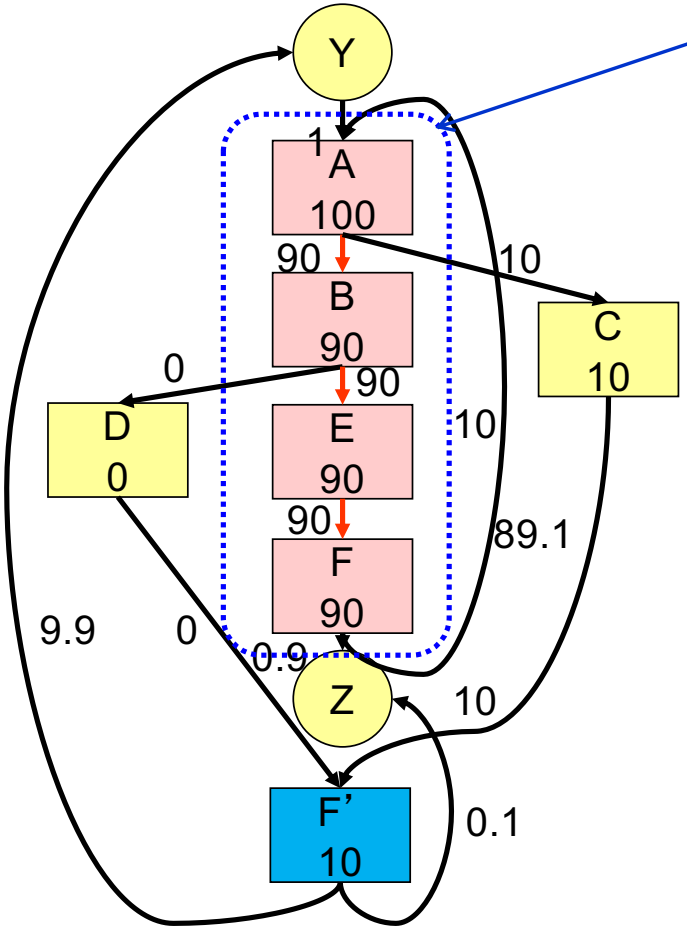


- Hwu et al. "The Superblock: An effective technique for VLIW and superscalar compilation," Journal of Supercomputing, 1993.

Superblock Formation (I)



Superblock Formation (II)



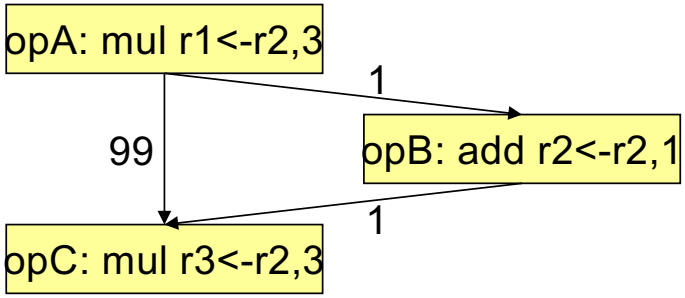
This is a **superblock**

(code with single entry point, multiple exit points)

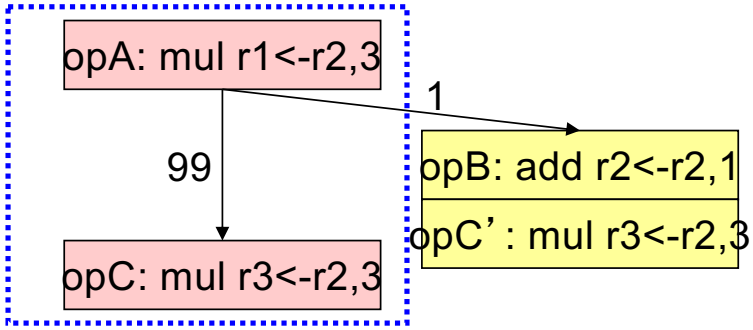
Tail duplication:
duplication of basic blocks after a side entrance to eliminate side entrances

→ transforms a trace into a superblock

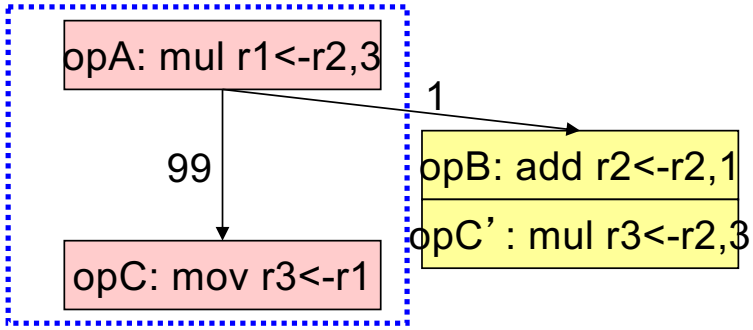
Superblock Code Optimization Example



Original Code



Code After Superblock Formation



Code After Common Subexpression Elimination

Paper on Superblock Formation

The Superblock: An Effective Technique

for VLIW and Superscalar Compilation

Wen-mei W. Hwu Scott A. Mahlke William Y. Chen Pohua P. Chang

Nancy J. Warter Roger A. Bringmann Roland G. Ouellette Richard E. Hank

Tokuzo Kiyohara Grant E. Haab John G. Holm Daniel M. Lavery *

Hwu et al., The superblock: An effective technique for VLIW and superscalar compilation.
The Journal of Supercomputing, 1993.

- Lecture Video on Static Instruction Scheduling
  - https://www.youtube.com/watch?v=isBEVkIjgGA

Another Example Work: IMPACT

IMPACT: An Architectural Framework for Multiple-Instruction-Issue Processors

Pohua P. Chang *Scott A. Mahlke* *William Y. Chen* *Nancy J. Warter* *Wen-mei W. Hwu*

Center for Reliable and High-Performance Computing
University of Illinois
Urbana, IL 61801

The performance of multiple-instruction-issue processors can be severely limited by the compiler's ability to generate efficient code for concurrent hardware. In the IMPACT project, we have developed IMPACT-I, a highly optimizing C compiler to exploit instruction level concurrency. The optimization capabilities of the IMPACT-I C compiler are summarized in this paper. Using the IMPACT-I C compiler, we ran experiments to analyze the performance of multiple-instruction-issue processors executing some important non-numerical programs. The multiple-instruction-issue processors achieve solid speedup over high-performance single-instruction-issue processors.

Chang et al., IMPACT: an architectural framework for multiple-instruction-issue processors. ISCA 1991.

Another Example Work: Hyperblock

Effective Compiler Support for Predicated Execution Using the Hyperblock

Scott A. Mahlke David C. Lin* William Y. Chen Richard E. Hank Roger A. Bringmann

Center for Reliable and High-Performance Computing
University of Illinois
Urbana-Champaign, IL 61801

- Lecture Video on Static Instruction Scheduling
  - https://www.youtube.com/watch?v=isBEVkIjgGA

Mahlke+, "Effective Compiler Support for Predicated Execution Using the Hyperblock," MICRO 1992.

The Bulldog VLIW Compiler

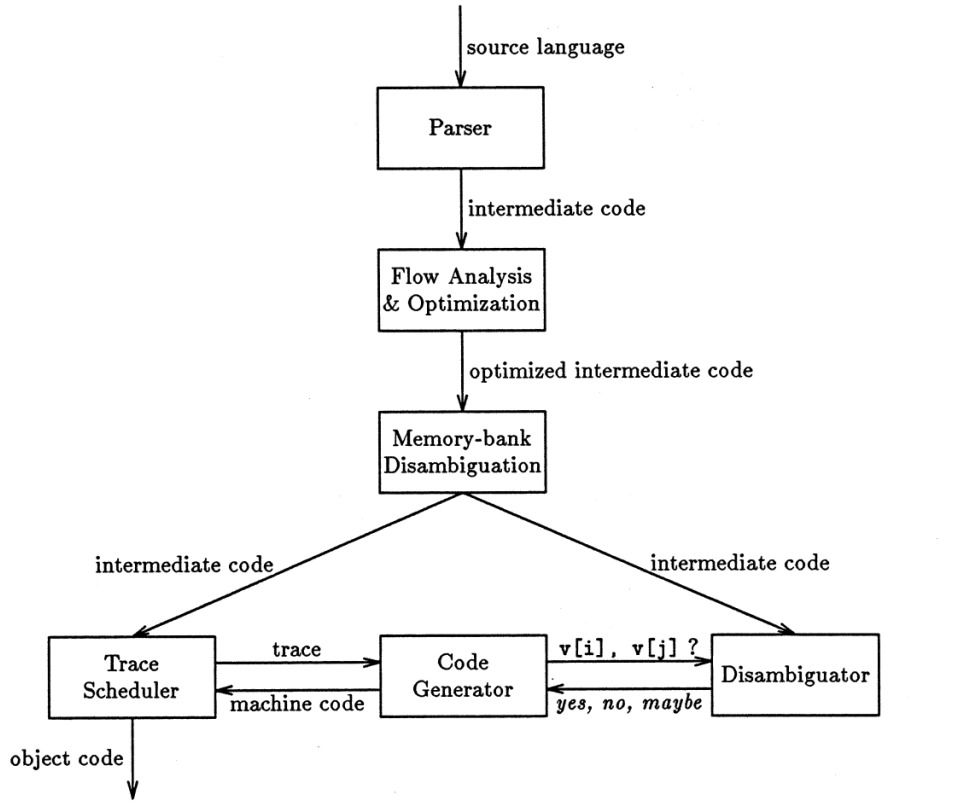
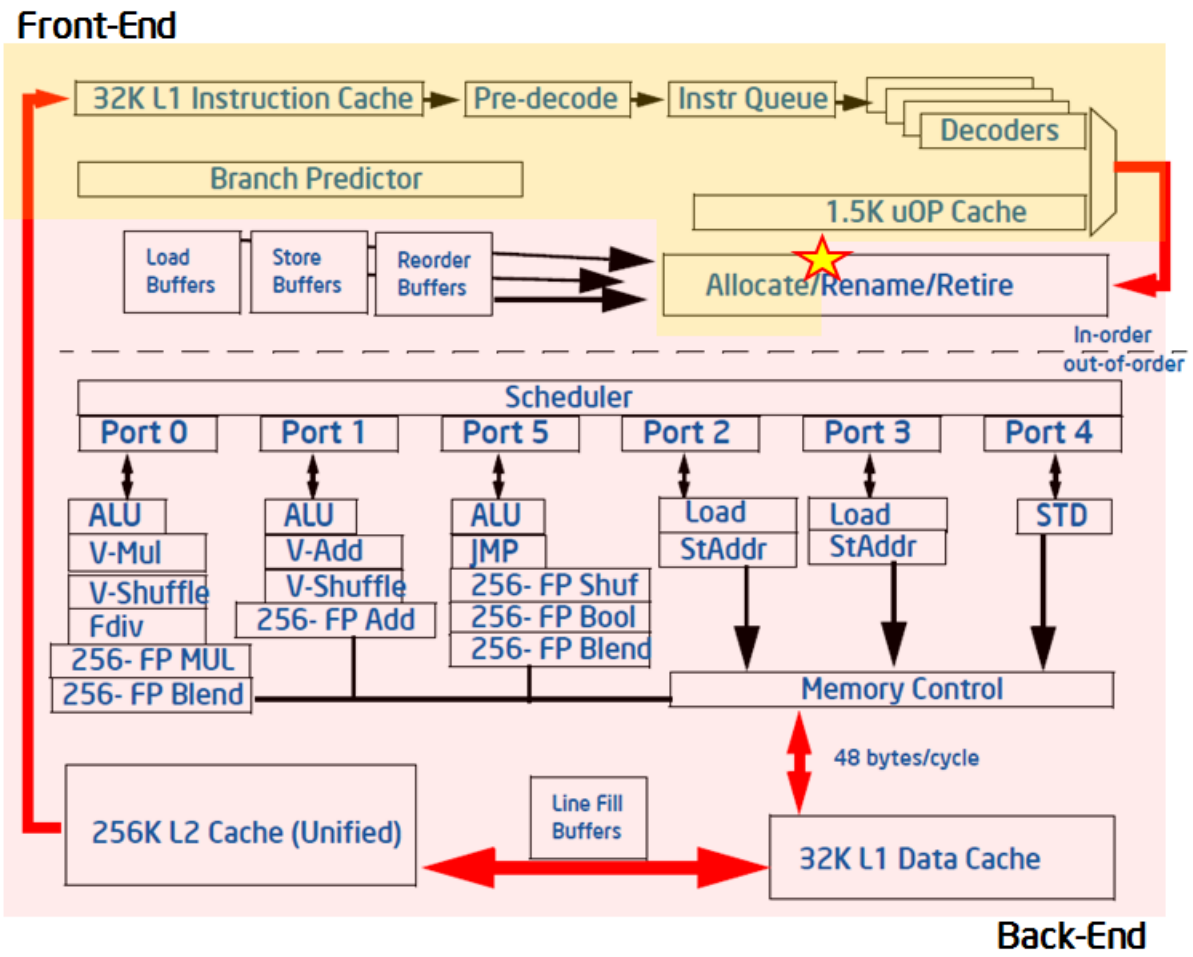


Figure 1.5. The Bulldog compiler.

John Ellis, "Bulldog: A Compiler for VLIW Architectures," PhD Thesis 1984.

It all helps software optimization



Analyzing and Improving the Scalability of In-Memory Indices for Managed Search Engines

Aditya Chilukuri
aditya.chilukuri@anu.edu.au
Australian National University
Canberra, ACT, Australia

Shoaib Akram
shoaib.akram@anu.edu.au
Australian National University
Canberra, ACT, Australia

Abstract

Managed search engines, such as Apache Solr and Elasticsearch, host huge inverted indices in main memory to offer fast response times. This practice faces two challenges. First, limited DRAM capacity necessitates search engines aggressively compress indices to reduce their storage footprint. Unfortunately, our analysis with a popular search library shows that compression slows down queries (on average) by up to 1.7× due to high decompression latency. Despite their performance advantage, uncompressed indices require 10× more memory capacity, making them impractical. Second, indices today reside off-heap, encouraging unsafe memory accesses and risking eviction from the page cache.

Emerging byte-addressable and scalable non-volatile memory (NVM) offers a good fit for storing uncompressed indices. Unfortunately, NVM exhibits high latency. We rigorously evaluate the performance of DRAM and NVM-backed compressed/uncompressed indices to find that an uncompressed index in a high-capacity managed heap memory-mapped over NVM provides a 36% reduction in query response times compared to a DRAM-backed compressed index in off-heap memory. Also, it is only 11% slower than the uncompressed index in a DRAM heap (fastest approach). DRAM and NVM-backed compressed (off-heap) indices behave similarly.

We analyze the narrow response time gap between DRAM and NVM-backed indices. We conclude that inverted indices demand massive memory capacity, but search algorithms exhibit a high spatial locality that modern cache hierarchies exploit to hide NVM latency. We show the scalability of uncompressed indices on the NVM-backed heap with large core counts and index sizes. This work uncovers new space-time tradeoffs in storing in-memory inverted indices.

ISMM '23, June 18, 2023, Orlando, FL, USA

ACM ISBN 979-8-4007-0179-5/23/06.
https://doi.org/10.1145/3591195.3595272

CCS Concepts: • **Information systems** → **Search index compression**; Search engine indexing; • **Hardware** → **Memory and dense storage**; • **Software and its engineering** → *Garbage collection*.

Keywords: Text search, inverted index, persistent memory, compression, managed heap, garbage collection

ACM Reference Format:
Aditya Chilukuri and Shoaib Akram. 2023. Analyzing and Improving the Scalability of In-Memory Indices for Managed Search Engines . In *Proceedings of the 2023 ACM SIGPLAN International Symposium on Memory Management (ISMM '23), June 18, 2023, Orlando, FL, USA.* ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3591195.3595272

1 Introduction

Search engines enable locating web pages on the internet and are a critical component of social media, professional networking, and e-commerce platforms. The key to retaining satisfied users is to offer low query response times. Amazon reports that even a 100 ms delay results in revenue drops [36]. Similar observations guide Google's search infrastructure.

The critical data structure search engines use for locating documents (web pages or social media posts) matching a word (term) is an inverted index. An inverted index maps unique terms to posting lists, where each posting stores an integer document identifier (ID) and meta-data (term frequency and position). Associating terms to posting lists using an inverted index speeds up query evaluation dramatically.

Today's standard practice is to host the inverted index in off-heap main memory. Recent work shows that even PCIe NVMe SSDs with byte-addressable 3D XPoint memory cannot deliver real-time response times [2]. Therefore, service providers keep indices in memory [60]. Unfortunately, as datasets grow, the inverted index grows proportionally, and large indices put increased pressure on DRAM. On the other hand, DRAM scaling cannot cope with the growth in datasets [20, 42]. Specifically, as data volume doubles yearly, the DRAM capacity only scales by 10% [24, 28]. The result is either the poor quality of service due to index lookups from storage or exorbitant memory-related expenditures.

Problem # 1 (High Decompression Latency): Compression is a crucial technique search engines use to store large indices in limited DRAM. For example, Apache Lucene uses a compression scheme that reduces index size by 85–90%,

Honors thesis *best paper candidate at ISMM*

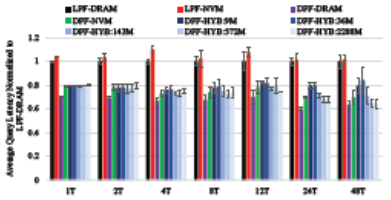


Figure 7. Showing average query latency normalized to the LPF-DRAM baseline for different systems with single-term queries. (T stands for thread count.)

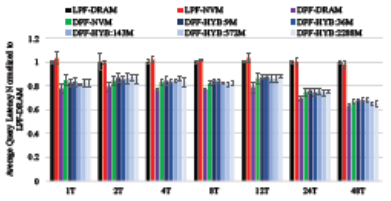


Figure 8. Showing average query latency normalized to the LPF-DRAM baseline for different systems with 2-term AND queries. (T stands for thread count.)

memory. On the other hand, DPF-NVM is slower (7%) than DPF-DRAM, the fastest approach today, i.e., hosting uncompressed indices in DRAM. We regain the lost performance with the intuition that most queries last only a few tens of milliseconds, but they allocate temporary, short-lived objects in the nursery. Queries incur a high latency if new object allocation happens in a slow media (NVM). Figure 7 shows that a hybrid system with a well-tuned nursery size (4× to 64×) is 6% faster than DPF-NVM (36% compared to LPF-DRAM), which places the entire heap on slow NVM. Our proposed hybrid system offers an attractive solution: ❶ it consumes between 0.5 GB to 2 GB DRAM, ❷ exploits the NVM capacity for hosting the uncompressed index, ❸ and is only 1% (48 cores) slower than DPF-DRAM. We observe that the best-performing nursery size varies across thread counts, and a hybrid system's performance benefits are more significant at high concurrency levels. This observation is in line with prior work that reports NVM bandwidth saturates rapidly beyond eight threads [3, 62], and hence the benefits of a hybrid approach that mitigates pressure on NVM by isolating new allocations in DRAM are higher. On average, small nurseries increase the GC overhead. Automatically tuning the DRAM nursery size is future work.

We observe similar behaviors for two-term conjunctive queries. One notable difference is that the gap between LPF-DRAM and DPF-DRAM is generally less wide, especially at low thread count. In general, conjunctive queries are more compute-intensive than single-term queries because of the large number of comparisons across multiple posting lists. At 48 threads, conjunctive queries with DPF-NVM, on average, suffer a 4% slowdown compared to the fastest system (DPF-DRAM). A hybrid system with a well-tuned nursery (15× the LLC size) bridges the gap to only 2% of DPF-DRAM.

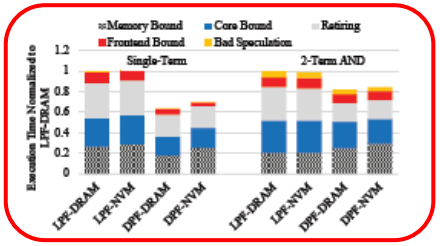


Figure 9. Showing the breakdown of per-query execution time into various components representing microarchitectural bottlenecks

4.3.2 Microarchitectural Analysis. We now report observations from our detailed microarchitectural analysis of query workloads. We aim to understand how search queries interact with a server's cache and memory hierarchy. We use the top-down methodology [66] that systematically identifies *true bottlenecks* in an out-of-order processor. It identifies bottlenecks by rigorous performance counter measurements. Figure 9 shows the results of our analysis for two query workloads and different memory systems (48 threads). We breakdown query execution times (normalized to LPF-DRAM) into five components: ❶ backend memory-bound due to long-latency memory operations (cache hits or misses), ❷ backend core-bound due to lack of core resources, such as functional unit or reservation station, ❸ smoothly retiring instructions, ❹ frontend bound due to, e.g., lack of decoded microps, and ❺ recovering from misspeculation. Unfortunately, we observe that queries spend a significant portion of the execution time resolving memory loads (high memory-bound component) due to the data-intensive nature of search workloads. ILP is low due to the dependencies between instructions performing binary searches and skip-list traversals. Mispredicted branches cost very few cycles.

We observe that the memory-bound portion of the execution time is the highest (up to 36%) for DPF-NVM, while it is 30% for other systems. Reading uncompressed indices stresses NVM's bandwidth, especially at high thread count.

Summer RA *submission to top-tier database conference*

SPIRIT: Scalable and Persistent On-Heap Indices in Hybrid Memory for Real-Time Search

Adnan Hasnat
Adnan.Hasnat@anu.edu.au
Australian National University
Canberra, ACT, Australia

Shoaib Akram
Shoaib.Akram@anu.edu.au
Australian National University
Canberra, ACT, Australia

ABSTRACT

Today, real-time search over big microblogging data requires low indexing and query latency. Online services, therefore, prefer to host inverted indices in memory. Unfortunately, as datasets grow, indices grow proportionally, and with limited DRAM scaling, the main memory faces high pressure. Also, indices must be persisted on disks as building them is computationally intensive. Consequently, it becomes necessary to frequently move on-heap index segments to block storage, slowing down indexing. Reading storage-resident index segments necessitates filesystem calls and disk accesses during query evaluation, leading to high and unpredictable tail latency.

This work exploits (hybrid) DRAM and scalable non-volatile memory (NVM) to offer dynamically growing, and instantly searchable, large persistent indices in on-heap memory. We implement our proposal in SPIRIT, a real-time text inversion engine over hybrid memory. SPIRIT exploits the byte-addressability of hybrid memory to enable direct access to the index on a pre-allocated heap, eliminating expensive block storage accesses and filesystem calls during live operation. It uses fast persistent pointers in a global descriptor table to offer: ❶ instant segment availability to query evaluators upon fresh ingestion, ❷ low-overhead segment movement across memory tiers transparent to query evaluators, and ❸ decoupled segment movement into NVM from their visibility to query evaluators, enabling intelligent policies for mitigating high NVM latency. SPIRIT accelerates compaction with zero-copy merging and supports fast, graceful shutdown and instant recovery.

With 15% and 50% of the index in DRAM, it concurrently resolves queries only 13% and 6.44% slower, respectively, compared to DRAM alone. Our work generalizes to other data-intensive services that will benefit from direct on-heap access to large persistent indices.

PVLDB Reference Format:
Adnan Hasnat and Shoaib Akram. SPIRIT: Scalable and Persistent On-Heap Indices in Hybrid Memory for Real-Time Search. PVLDB, 14(1): XXX-XXX, 2020.
doi:XX.XX/XXX.XX

PVLDB Artifact Availability:
The source code, data, and/or other artifacts have been made available at https://github.com/vics1/SPIRIT.

Proceedings of the VLDB Endowment, Vol. 14, No. 1 ISSN 2150-8097.
doi:XX.XX/XXX.XX

1 INTRODUCTION

Today, enabling fast real-time search over social media content is critical to the success of many enterprises, including Linkedin, Meta, and Twitter [8, 52]. Social media content is either queried explicitly for *relevance search*, similar to static web content, or implicitly by the service for *timeline retrieval* to populate a user's home feed. The latter generates queries based on the user's preferred topics or followers and is more frequent on social media platforms — the source of 50% of tweets recommended on Twitter's `For You` and `Following` tabs [69]. The queries run concurrently with an indexing engine that builds indices in real-time, coping with a massive volume of data, e.g., 500 million Tweets per day for Twitter [69]. Consequently, real-time indexing puts high pressure on the aggregate CPU and memory of the real-time cluster, and concurrent queries exacerbate the pressure further [3, 52].

The critical data structure search engines use for locating documents (web pages, social media posts, or tweets) matching a word (term) is an inverted index [84]. Offering real-time response times requires hosting the index in memory [28, 75]. Unfortunately, indices grow proportional to datasets, and large indices put pressure on DRAM. However, DRAM scaling cannot cope with the growth in datasets [1, 11, 14, 28, 29, 33, 57, 61, 72], increasing infrastructure cost [15, 72]. Scaling in-memory indices to large datasets demands dense memory technologies, complementing DRAM.

Hosting indices in memory is also at odds with persisting them in today's storage stack. Popular search engines, e.g., Apache Solr [22] and Elasticsearch [20] use a segmented index, and each fixed-size segment resides on the heap before being moved to the page cache. Segments are buffered in the cache (no synchronous I/O per segment) to amortize I/O overheads. Ultimately, calling `fsync` to bulk-persist segments on storage, an operation called *commit* in Elasticsearch, becomes necessary to avoid losing significant index updates on a crash [5, 20]. Unfortunately, if the index outgrows available DRAM, later reads of the persistent segment generate I/O transfers. Recent work shows that even the fastest PCIe NVMe SSDs cannot deliver the response times required by real-time search [3, 52, 69]. Furthermore, an `fsync` is costly and cannot be performed without a significant performance hit [20]. Existing systems either risk losing a significant state or paying a performance penalty.

Due to limited DRAM and the need for persistence, block storage is deeply integrated into real-time search clusters. Therefore, popular real-time search engines do not make on-heap segments visible to query evaluators [20]. Instead, query evaluators use multiple filesystem calls to access new segments from the OS cache or storage. (In Elastic search, one set of calls is to read the commit point and another to read the index segment [20].) These calls incur high overhead [9, 38], prohibiting real-time operation [20]. Furthermore, to keep busy during the time it takes for the indexer to process 1 M documents. In WRT, we start our QPS measurement after a threshold of documents are ingested. The intuition for WRT is that initially the index is empty and all queries, popular and others, are resolved instantly, inflating QPS.

Query Formation. We use two query types: ❶ single-word (ST) and ❷ multi-word (MT) conjunctive queries. Our workloads are homogeneous, and we do not mix S and M queries. We use terms from `topTerms20120502.txt` available on the Luceneutil website. The terms are divided into low, medium, and high categories. We form a query workload suite consisting of six workloads: L, M, H, LL, MM, and HH. We validate critical findings for workloads with up to 100 K queries. We use a query thread pool for resolving queries, and a single query is resolved sequentially.

Default Parameters. Unless otherwise stated, we use the following default parameters. We index 1 M documents, measure QPS for 100 K queries, and use six query workloads. We vary the number of executors from 1 to 16, and use one query thread per executor. The segment size is 128 MB. The ephemeral and long-lived, DRAM and NVM, heaps are 1 GB, 5 GB, 5 GB, and 20 GB, respectively. We use the naive advance algorithm for intersection queries, and the hash table DRAM-resident dictionary. In WRT mode, the window is sized as follows: first the indexer ingests 800 K documents, and the QPS is measured while the queries execute concurrently with ingestion of 200 K additional documents.

Statistical Significance Our results are statistically significant. The coefficient of variation (COV) for (average) indexing time across eight experimental runs is 0.25%, 0.69%, and 0.38% for 1, 4, and 16 executors. In RT and WRT modes, the variation is higher, and the COV is up to 1.6%. For QPS, across hundreds of configurations, the COV is between 0.09% (HH queries, one executor) and 2.6% (M queries, 16 executors).

7 EVALUATION RESULTS

Using DRAM alone in a real-time cluster increases DRAM cost, but it is necessary today to avoid storage latency. *Our evaluation aims to establish if we can use NVM to mitigate DRAM pressure and still deliver real-time operation.* We discuss SPIRIT's performance in non-concurrent and real-time modes, varying DRAM size. We also perform several sensitivity studies (e.g., dictionary type, segment size, intersection algorithm, and NVM bandwidth). More specifically, we ask the following key questions.

- How much slower is NVM-Only compared to DRAM-Only in non-concurrent mode? Does merging bridge this gap?
- What is the impact of DRAM size (as a percentage of index size) on SPIRIT's indexing and query latency?
- How does SPIRIT's indexing pipeline behave, and what is the impact of concurrent queries?
- What is the overhead of graceful shutdown and recovery?

7.1 Query Performance

We first report the QPS for DRAM-Only and NVM-Only in NC mode with and without merging, varying the executor count. We then show the impact of heap sizing on QPS in WRT mode.

DRAM-Only versus NVM-Only. We compare QPS with DRAM-Only and NVM-Only. For a fair comparison against DRAM-Only,

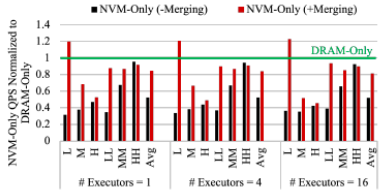


Figure 2: Showing the QPS with NVM-Only relative to DRAM-Only for single-term and multi-term query workloads.

we first disable merging in NVM, placing non-merged segments in DRAM or NVM. We execute queries after indexing and show the results of our evaluation in Figure 2. The figure shows the QPS with NVM-Only against a DRAM-Only baseline. (DRAM-Only is one, and higher is better.) We first focus on results without merging. We observe that across a different number of executors, on average, across all workloads, NVM-Only is 48% slower than DRAM-Only. Specifically, L queries are up to 68% slower with NVM-Only. The DRAM-NVM gap opens up with single-term and MM queries. The HH queries are less sensitive to NVM latency (only 4% slower with NVM). We observe a trend: as the CPU overhead per memory access increases, the query workload is less sensitive to memory latency. Most of the time, L queries are so unpopular that they do not even traverse the posting lists and merely perform a dictionary lookup. Therefore, a single lookup is slower in NVM than in DRAM. At the other end of the extreme, the HH queries perform a compute-intensive intersection operation, and therefore, for each index (memory) access, they incur a substantial CPU overhead. The MM (32% slower) queries lie between the two extremes.

Impact of merging. We now discuss the normalized QPS of NVM-Only with merging enabled. Many non-merged segments increase the dictionary lookup latency because each non-merged segment is an independent index. With 16 executors, on average, across six query workloads, QPS with merging is 33% higher than no merging. We observe two surprising results in Figure 2. First, the QPS of L queries with NVM-Only is even better than DRAM-Only (up to 23%). This better QPS is because the cost of many hash table lookups slows down L queries much more than the slow access latency of NVM. Second, HH queries are slower (4%) with merging compared to no merging. For HH queries, the cost of a dictionary lookup is only a fraction of the total query latency. Therefore, searching over a merged segment does not speed up HH queries. The slightly worst performance with merging is likely due to locality effects.

Heap sensitivity. Figure 3 shows the QPS of SPIRIT in WRT mode for different heap configurations. Specifically, we vary the long-term DRAM heap and observe the QPS. The long-term DRAM impacts QPS as SPIRIT retains segments in DRAM upon engraving and only makes them visible to query evaluators when it runs out of DRAM. The more the DRAM capacity available to SPIRIT, the better the QPS, especially for L, M, LL, and MM query workloads. To verify this hypothesis, we experiment with three heap configurations and size them relative to the total size of the index. The expectation is

PhD student's work *ASPLOS (super selective)*

TeraHeap: Reducing Memory Pressure in Managed Big Data Frameworks

Iacovos G. Kolokasis*† — FORTH-ICS, Greece — kolokasis@ics.forth.gr
Giannos Evdorou*† — FORTH-ICS, Greece — evdorou@ics.forth.gr
Shoaib Akram‡ — ANU, Australia — shoaib.akram@anu.edu.au
Christos Kozanitis* — FORTH-ICS, Greece — kozanitis@ics.forth.gr
Anastasios Papagiannis — Isovelent, Inc., USA — anastasios@isovalent.com
Foivos S. Zakkak — Red Hat, Inc., UK — fzakkak@redhat.com
Polyvios Pratikakis*† — FORTH-ICS, Greece — polyvios@ics.forth.gr
Angelos Bilas*† — FORTH-ICS, Greece — bilas@ics.forth.gr

ABSTRACT

Big data analytics frameworks, such as Spark and Giraph, need to process and cache massive amounts of data that do not always fit on the managed heap. Therefore, frameworks temporarily move long-lived objects outside the managed heap (off-heap) on a fast storage device. However, this practice results in (1) high serialization/deserialization (S/D) cost and (2) high memory pressure when off-heap objects are moved back to the heap for processing.

In this paper, we propose *TeraHeap*, a system that eliminates S/D overhead and expensive GC scans for a large portion of the objects in big data frameworks. *TeraHeap* relies on three concepts. (1) It eliminates S/D cost by extending the managed runtime (JVM) to use a second high-capacity heap (H2) over a fast storage device. (2) It offers a simple hint-based interface, allowing big data analytics frameworks to leverage knowledge about objects to populate H2. (3) It reduces GC cost by fencing the garbage collector from scanning H2 objects while maintaining the illusion of a single managed heap.

We implement *TeraHeap* in OpenJDK and evaluate it with 15 widely used applications in two real-world big data frameworks, Spark and Giraph. Our evaluation shows that for the same DRAM size, *TeraHeap* improves performance by up to 73% and 28% compared to native Spark and Giraph, respectively. Also, it provides better performance by consuming up to 4.6× and 1.2× less DRAM capacity than native Spark and Giraph, respectively. Finally, it outperforms Panthera, a state-of-the-art garbage collector for hybrid memories, by up to 69%.

CCS CONCEPTS

• **Software and its engineering** → **Memory management**; **Garbage collection**; **Runtime environments**; • **Information systems** → **Flash memory**; **Phase change memory**; *Data analytics*; • **Computer systems organization** → **Cloud computing**.

KEYWORDS

Java Virtual Machine (JVM), large analytics datasets, serialization, large managed heaps, memory management, garbage collection, memory hierarchy, fast storage devices

ACM Reference Format:
Iacovos G. Kolokasis, Giannos Evdorou, Shoaib Akram, Christos Kozanitis, Anastasios Papagiannis, Foivos S. Zakkak, Polyvios Pratikakis, and Angelos Bilas. 2023. TeraHeap: Reducing Memory Pressure in Managed Big Data Frameworks. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3 (ASPLOS '23), March 25–29, 2023, Vancouver, BC, Canada.* ACM, New York, NY, USA, 16 pages. https://doi.org/10.1145/3582016.3582045

*Foundation for Research and Technology - Hellas (FORTH), Institute of Computer Science (ICS), Greece
†Department of Computer Science, University of Crete, Greece
‡Australian National University, Australia

ASPLOS '23, March 25–29, 2023, Vancouver, BC, Canada

ACM ISBN 978-1-4503-9918-0/23/03.
https://doi.org/10.1145/3582016.3582045

1 INTRODUCTION

Managed big data frameworks, such as Spark [56] and Giraph [44], are designed to analyze huge volumes of data. Typically, such processing requires iterative computations over data until a convergence condition is satisfied. Each iteration produces new transformations over data, generating a massive volume of objects spanning long computations.

Hosting a large volume of objects on the managed heap increases memory pressure, resulting in frequent garbage collection (GC) cycles with low yield. Each GC cycle reclaims little space because (1) the cumulative volume of allocated objects is several times larger than the size of available heap [51] and (2) objects in big data frameworks exhibit long lifetimes [10, 48, 52]. Although production garbage collectors efficiently manage short-lived objects, they do not perform well under high memory pressure introduced by long-lived objects [32].

The common practice for coping with rapidly growing datasets and high GC cost is to move objects outside the managed heap (off-heap) over a fast storage device (e.g., NVMe SSD). However, frameworks cannot compute directly over off-heap objects, and thus, they (re)allocate these objects on the managed heap to process them. Although some systems support off-heap computation over byte arrays with primitive types [14], they do not offer support for computation over arbitrary objects, resulting in applications specific solutions, such as Spark SQL [12].

Moving managed objects off-heap has two main limitations. First, it introduces high serialization/deserialization (S/D) overhead for applications that use complex data structures [34, 43, 50]. Recent

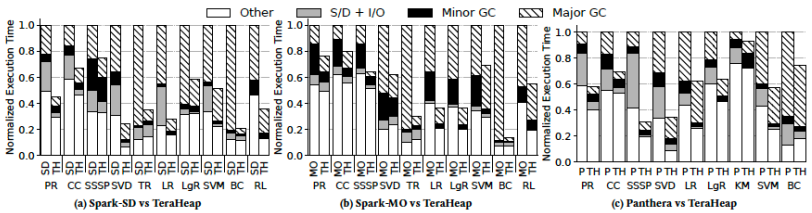


Figure 12: TeraHeap (TH) performance compared to (a) Spark-SD, (b) Spark-MO, and (c) Panthera (P) over NVM server.

TeraHeap when using NVM to increase the heap size, which can eliminate S/D at increased GC cost for native. Figure 12(a) shows that *TeraHeap* improves performance by up to 79% and on average by 56%, compared to Spark-SD. Unlike the off-heap cache in Spark-SD, *TeraHeap* allows Spark to directly access cached objects in H2 via load/store operations to NVM, without the need to perform S/D. *TeraHeap* significantly reduces S/D and GC time compared to Spark-SD by up to 97% and 93%, respectively.

Figure 12(b) shows that *TeraHeap* improves performance by up to 86% and on average by 48%, compared to Spark-MO. The main improvement of *TeraHeap* results from the reduction of minor GC and major GC time by up to 88% (on average by 52%) and 96% (on average by 46%) compared to Spark-MO, respectively. In Spark-MO, running the garbage collector on top of NVM (using DRAM as a cache) incurs high overhead due to the latency of NVM [53] and the agnostic placement of objects. For instance, minor GC time in Spark-MO increases on average by 36% compared to Spark-SD (Figure 12b) because objects of the young generation are placed in NVM, resulting in higher access latency for the garbage collector. Unlike *TeraHeap* that controls object placement in NVM (H2), Spark-MO relies on the memory controller to move objects between DRAM and NVM. We measure that Spark-MO incurs on average 5.3× and 11.8× more read and write operations to NVM compared to *TeraHeap*, resulting in higher overhead. Therefore, the ability to maintain separate heaps allows *TeraHeap* to both limit GC cost and reduce the adverse impact of the increased NVM access latency on GC time.

We also compare *TeraHeap* with Panthera [48][1], a system designed to use NVM as a heap in Spark. Panthera extends the managed heap over DRAM and NVM, placing the young generation in DRAM and splitting the old generation into DRAM and NVM components. We configure Panthera similar to Wang *et. al* [48] with 64 GB heap, 25% on DRAM (16 GB), and 75% on NVM. We set the size of the young generation to $\frac{1}{6}$ (10 GB) of the total heap size and place it entirely on DRAM. We set the size of the old generation to the rest of the heap size (54 GB) and place 6 GB on DRAM and the rest (48 GB) on NVM. We configure *TeraHeap* to use an H1 of 16 GB and map H2 to NVM. Thus, both systems use the same DRAM and NVM capacity.

[1]As Panthera is not publicly available, we are thankful to the authors for providing us their code.

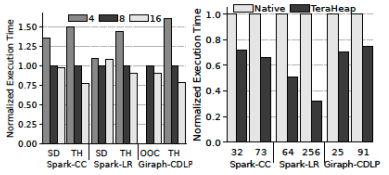


Figure 13: Performance scaling with (a) number of mutator threads and (b) dataset size in the NVMe server.

Figure 12(c) shows that *TeraHeap* improves performance between 7% and 69% compared to Panthera across all workloads. Panthera bypasses the allocation of some objects in the young generation, allocating them directly to the old generation. However, each major GC still scans all objects in the old generation, which increases overhead as the heap address space grows. Instead, *TeraHeap* reduces the address space that needs to be scanned by the garbage collector. Note that Panthera incurs more accesses to NVM because it allocates mature long-lived objects that are highly read and updated by the mutator threads. Specifically, it increases other by up to 53% because it performs more NVM read (up to 54×) and NVM write (up to 51×) operations than *TeraHeap*.

7.6 Performance Scaling

A benefit of *TeraHeap* is that it allows increasing the number of mutator threads in Spark and Giraph executors. In both Spark and Giraph, each mutator thread processes a separate partition. Thus, as the number of threads in the executor increases, the object allocation rate increases, leading to higher GC cost. Figure 13(a) shows the performance of CC, LR, and CDLP (other workloads show similar behavior) using Spark-SD, Giraph-OOC, and *TeraHeap* (TH) with 4, 8, and 16 threads, normalized to 8 threads per configuration. We note that Giraph-OOC with four threads results in an OOM error. *TeraHeap* allows applications to scale performance further to 23% with 2× more threads. However, Spark-SD does not scale beyond 8 threads in LR because GC cost increases (by 44%), eliminating

MSQuest: An FPGA Accelerator for Peptide Database Search

PhD student's work *Bioinformatics journal*

Abstract—In proteomic and peptidomic analyses, mass-spectrometry (MS) based peptide identification is at the heart of cancer bio-marker studies. The most popular approach to deduce peptides from MS data are database search algorithms which operate by matching mass spectra of biological samples against a large database of theoretical peptide sequences. This process is time-consuming and requires too many computational resources specially when post-translational modifications (PTMs) are incorporated in the theoretical database. We present an FPGA-based accelerator, called MSQuest to accelerate database search process using a hardware/software co-design methodology. First, we theoretically analyzed the algorithm to reveal parallelization opportunities and computational bottlenecks. Second, we designed an architectural template for the FPGA to exploit different sources of parallelism inherent in the computational workload. Third, we formulated an analytical performance model for the architecture template to perform design space exploration (DSE) and find a near-optimal architectural configuration. Finally, we implemented our design on Intel Stratix 10 FPGA platform and validated it using real-world proteomics datasets and peptide database search experiments.

Index Terms—Mass-Spectrometry, FPGA, HW/SW co-design, database search, proteomics, accelerator

1 Introduction

Mass spectrometry based analysis is the foundation of large scale proteomics, peptidomics and proteogenomics studies [1], [2]. The discovery of novel disease biomarkers by MS analysis enables early detection of tumors, determine treatment and prognosis, and provide a deeper understanding of disease pathology which is the basis for developing personalized and precision medicine. Incorporating proteomics profiling in a clinical setting is an active goal of systems biology researchers [3]. Recent advancements in MS instrumentation techniques has greatly improved the quality of spectra that can be generated from complex clinical samples which is crucial in discovering clinically relevant biomarkers [4]. Thus, MS based peptide identification in proteomics and peptidomics is set to be the driving force behind diagnosing and treating genetic disorders via precision medicine [3].

To date, database search is the most popular approach to identify proteins from mass-spectrometry data [5], [6]. In database search, the first step is to generate a database containing all possible peptides which can result from enzymatic digestion of the protein sequences. To find a match in the generated database for each experimental spectrum, a similarity score is computed to quantitatively compare the mass spectrum against the set of candidate peptide sequences filtered by their precursor masses. In this process, the accuracy of a match is affected by two factors: quality of experimental spectra, and the presence of the corresponding peptide in the filtered candidates [7]. Thus, if the sample contains known peptides they are easily found, but most peptides go through post-translational modifications (PTMs) which are not accounted for in the theoretical database. Possible solutions to this include performing an unrestricted search by relaxing the mass filter or incorporating all possible PTMs in the search process when constructing the peptide database. However, an unconstrained search and inclusion of PTMs results in a combinatorial increase in the database size leading to a gigantic search space that requires too much time and computational resources [8], [9]. The current state-of-the-art database search frameworks lead to impractical search times (several days to weeks) when searching for PTMs [10].

Today, most popular database search frameworks that run on commodity computers are SEQUEST [11], Crux [12], Comet [13], X!tandem [14], and MSFragger [15]. To accelerate database search process, many researchers have utilized specialized computing platforms such as multicore [15], [16], [17] and distributed-memory architectures (HPC) [10], [18], [19], [20], [21], [22], graphical processing units (GPU) [23], [24], [25], [26], [27], and field programmable gate arrays (FPGA) [28], [29], [30].

In this work, we propose MSQuest, a novel FPGA-based hardware/software co-design approach to efficiently accelerate protein database search. We begin with a theoretical analysis of existing database search algorithms to identify different sources of parallelism at each stage of computation. Informed by our theoretical analysis, we design a processing element(PE) to compute similarity score that is configurable based on loop unroll factor. Moreover, to exploit all sources of parallelism, our overall architecture template allows loop pipelining and loop tiling to be configurable as well. Next, to find optimal design parameters, we derive an analytical performance model for our architecture template that takes into account the resource budget of the FPGA. Finally, we implement the design on Intel Stratix 10 FPGA and evaluate the design extensively with real world datasets to show improvement in runtime. The main contributions of this paper are:

- Design of a parameterized processing element with configurable loop unroll factor
- A novel architecture template design guided by theoretical analysis of the database search computation along with performance model to find optimal design configuration parameters

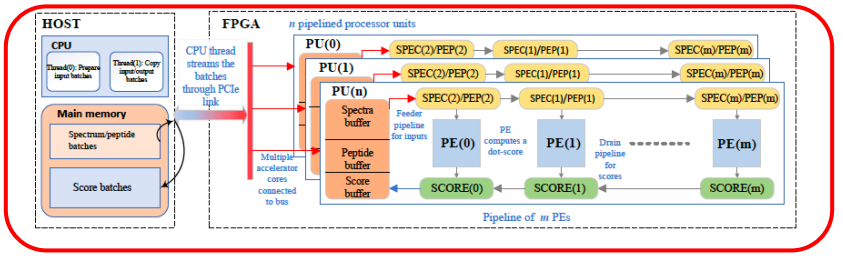


Fig. 2: Highlevel organization of the accelerator system. Host CPU communicates with the FPGA via a PCIe link. All n pipelined processing units (PUs) are exposed to the CPU host. CPU periodically polls the PUs, and copies the respective spectra, peptides, or scores. Each PU is composed of a pipeline of m PEs through which peptides and scores are streamed.

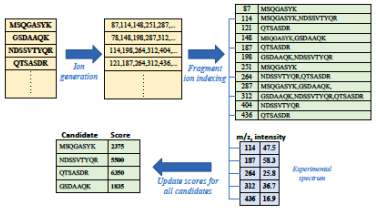


Fig. 3: Fragment-ion index generation procedure. First all the theoretical spectra are computed from the peptides, then the spectra are indexed where each fragment-ion is a key and the peptides that share the ion are values.

Algorithm 2 Compute with fragment-ion index

Require: M experimental spectra with top 150 peaks each
Require: On average k candidate peptides per spectrum
Ensure: Pre-computed fragment-ion indexed database
▷ m is a vector of mz values, I is a vector of candidate peptides, S is score matrix
1: **for** m ← 1 to M **do**
2: **for** i ← 1 to 150 **do**
3: v ← $bin[m[i]]$
4: **for** n ← 1 to k **do**
5: $S[m][v[n]] \leftarrow S[m][v[n]] + I[m[i]]I[v[n]]$
6: **end for**
7: **end for**
8: **end for**

2.5 Parallelism analysis

There are several sources of parallelism in database search process. To efficiently accelerate this computation, all of these sources must be correctly identified and exploited. Algorithm 1 and algorithm 2 show the psuedocode for database search without indexing and with indexing, respectively. The overall computation is represented by three nested for-loops:

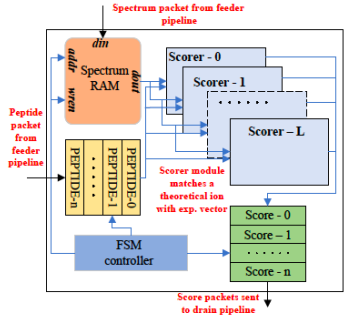


Fig. 4: Hardware design of the configurable processing element. An FSM controller reads the spectrum packets and peptide packets from the pipeline, and stores them in local on-chip RAM and shift registers. The scorer modules compute the dot-product concurrently, and the number of scorers is configurable according to the loop-unroll factor.

- **Inner loop:** In index-free approach, lines 4-7 compute the dot-product score between an experimental vector and a theoretical vector by iterating over 150 experimental spectrum peaks and performing the inner-join operation with the theoretical vector. This step can be parallelized by unrolling the loop on a vector processing hardware.
  In indexed search approach, the inner loop updates the score between experimental spectrum and all