

COMP2300-COMP6300-ENGN2219

Computer Organization & Program Execution

Convener: Shoaib Akram
shoaib.akram@anu.edu.au



Australian
National
University

Instruction-Level Parallelism (ILP)

Instruction-Level Parallelism (ILP)

- Overlapping the execution of instructions is called instruction-level parallelism
- We have seen ILP by *in-order* pipelines
 - Scalar pipeline (**partial overlap**)
 - Superscalar pipeline (**full overlap**)
- Full overlap requires multiple functional units and datapaths
- ILP processors bet that many instructions in the program are independent of each other
- **We will now see more aggressive ILP exploitation**

Dependences and Hazards

- Uncovering ILP aggressively requires understanding dependences and hazards
- **Recall:** Dependence is a program's property
- **Recall:** Hazard is a microarchitecture property
 - We have seen
 - Read-after write hazard (due to true dependences)
 - Control hazard (due to branches in programs)

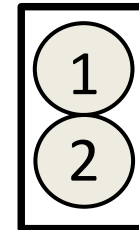
True dependence



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


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

- We have a single (dependent) instruction chain

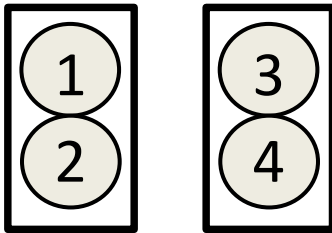


True dependences

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




- Two independent instruction chains



Anti dependence

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



- Anti dependence
 - 2 needs to write what 1 needs to read
- “Fake” or “false” dependence
 - R6 in **2** can be **replaced by (renamed to)** R7 to **eliminate** the (false) dependency

Output dependence

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

- Output dependence
 - 1 and 2 wants to write to the same register
- Again, it is a false dependence
 - We can **rename** register R2 in **2** to eliminate the dependence

Dependences and Hazards

- True dependence results in
 - Read-after-write hazard (RAW)
- Anti-dependence results in
 - Write-after-read hazard (WAR)
- Output dependence results in
 - Write-after-write hazard (WAW)
- **Single-cycle CPU**
 - None of the dependences result in a hazard (no concurrent execution)
- **Basic ARM 5-stage in-order pipeline**
 - Multiple instructions in different stages (possibility of RAW)
- **The CPUs we study today will exhibit WAR and WAW hazards**

From In-Order to Out-of-Order

In-Order Pipeline

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

ARM In-Order 5-Stage Pipe + Cache

- Assume ARM 5-stage pipeline with a cache and realistic memory (100 cycles)
- It has no option but to stall on a cache miss due to **structural dependency**
 - One ALU capable of executing “one operation” at a time
 - Memory can only handle “one request” at a time
 - If the MEM stage stalls, then EX stalls, and the entire pipeline stalls
- We will use a different more “**aggressive**” in-order pipeline to understand problem with in-order pipelines
 - Many **ALUs** and memory capable of handling **multiple requests** at a time
 - Eliminating the structural dependency and rearranging stages
 - Fetches, decodes, and issues one instruction **in order** every cycle to the functional units, but many of them can be in execution in any cycle

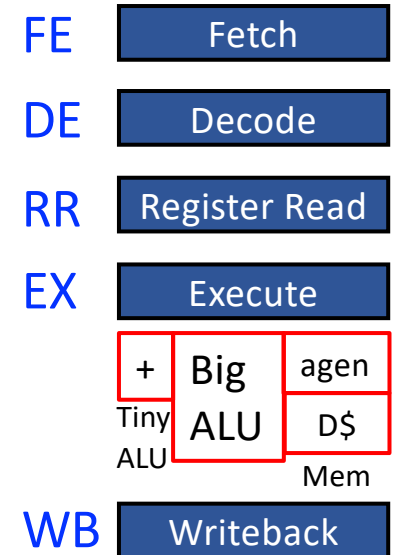
Stall-on-Use Pipeline

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



Stall-on-Use Pipeline

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

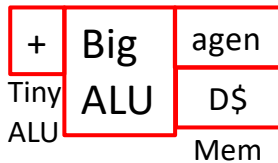


Fetch

Decode

Register Read

Execute



Writeback



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

Scenario 1: Load miss followed by independent instructions



Scenario 1: load miss followed by independent instructions

```

i1: LDR R2, [R1, #0]
i2: ADD R4, [R3, #1]
i3: ADD R6, R5, #2
i4: ADD R7, R6, #3
  
```

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



Scenario 1: load miss followed by independent instructions

```

i1: LDR R2, [R1, #0]
i2: ADD R4, [R3, #1]
i3: ADD R6, R5, #2
i4: ADD R7, R6, #3
  
```

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



Scenario 1: load miss followed by independent instructions

```

i1: LDR R2, [R1, #0]
i2: ADD R4, [R3, #1]
i3: ADD R6, R5, #2
i4: ADD R7, R6, #3
  
```

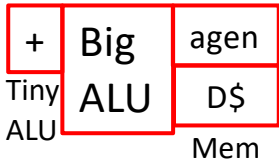
	1	2	3	4	5	6	7	8	9	10	11	12	13
i1	FE	DE	RR	EX _@	EX _{D\$}	...miss...							
i2		FE	DE	RR	EX	WB							
i3			FE	DE	RR	EX	WB						
i4				FE	DE	RR	EX	WB					

Fetch

Decode

Register Read

Execute



Writeback

i1

Scenario 1: load miss followed by independent instructions

i1: LDR R2, [R1, #0]

i2: ADD R4, [R3, #1]

i3: ADD R6, R5, #2

i4: ADD R7, R6, #3

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

Fetch

Decode

Register Read

Execute



Writeback

i1

Scenario 1: load miss followed by independent instructions

i1: LDR R2, [R1, #0]

i2: ADD R4, [R3, #1]

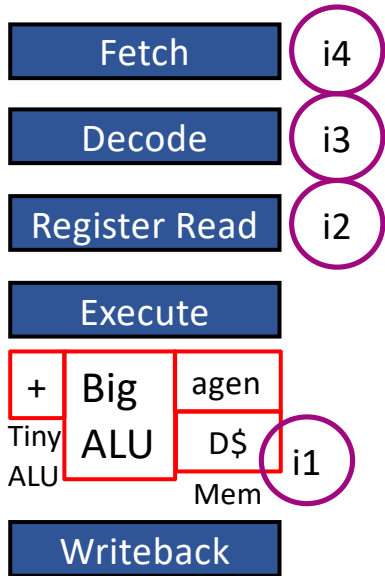
i3: ADD R6, R5, #2

i4: ADD R7, R6, #3

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

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

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



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

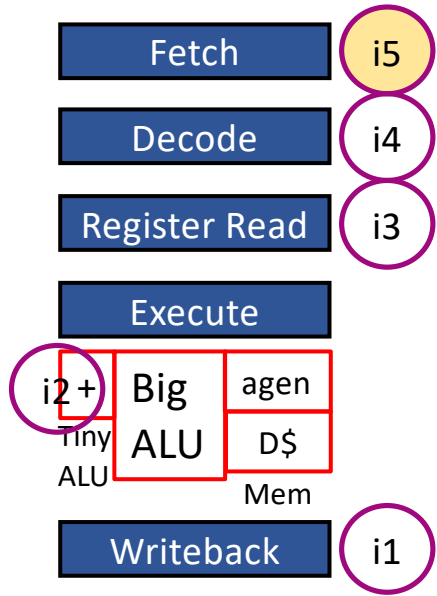
```

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

```

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

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

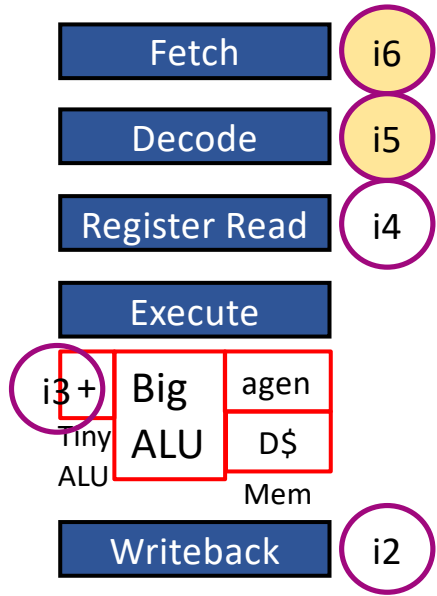


Scenario 2: load miss followed by dependent instruction, followed by independent instructions

```

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

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

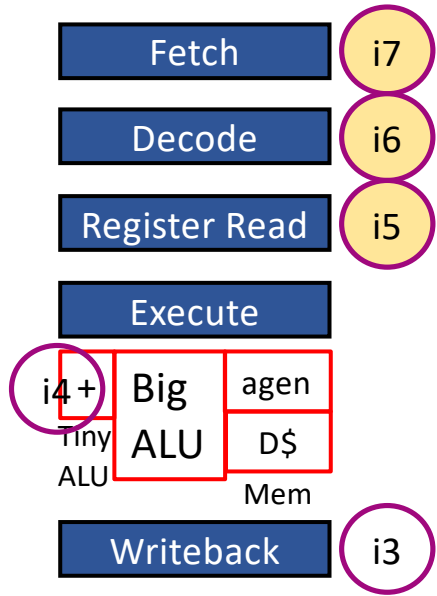


Scenario 2: load miss followed by dependent instruction, followed by independent instructions

```

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

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



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

```

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

Note: Blue arrows in the original image point from R2 in i1 to R2 in i2, and from R6 in i3 to R6 in i4.

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

Fetch

Decode

Register Read

Execute



Writeback

i4

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

i1: LDR R2, [R1, #0]

i2: ADD R4, R2, #1

i3: ADD R6, R5, #2

i4: ADD R7, R6, #3

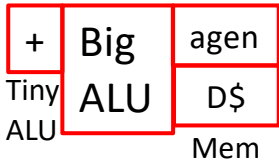
	1	2	3	4	5	6	7	8	9	10	11	12	13
i1	FE	DE	RR	EX _@	EX _{D\$}	...miss...				WB			
i2		FE	DE	RR	RR	RR	RR	RR	RR	EX	WB		
i3			FE	DE	DE	DE	DE	DE	DE	RR	EX	WB	
i4				FE	FE	FE	FE	FE	FE	DE	RR	EX	WB

Fetch

Decode

Register Read

Execute



Writeback

i4

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

i1: LDR R2, [R1, #0]

i2: ADD R4, R2, #1

i3: ADD R6, R5, #2

i4: ADD R7, R6, #3

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

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

What have we established?

- i_2 must wait for i_1
 - i_2 depends on i_1
- i_3, i_4 do not need to wait for the i_1-i_2 dependent chain (they are independent)
- But the i_3-i_4 chain stalls
 - Key insight: **In-order issue translates into a structural hazard**
 - RR stage (issue stage) blocked by the stalled i_2
- **In-order issue policy is the problem**
 - If a younger instruction has a **RAW** hazard with an older instruction (**must stall and it's ok!**)
 - What about instructions after it?
 - **Some of the younger instructions may be independent**
 - **This is where the problem lies!**

Nature of OOO

- **Out of order pipeline**
 - An instruction stalls if it has a **RAW hazard** with a previous instruction (that's ok!)
 - Independent instructions after it do not stall: **they may issue out of program order (OOO)**

Issue Queue

- OOO pipeline unblocks RR (issue) using a new **instruction buffer** for stalled data-dependent instructions
 - It is called “**reservation stations**”, “**issue buffer**”, “**issue queue**”, “**scheduler**”, “**scheduling window**”
- Stalled instructions do not impede instruction fetch
- Younger **ready** instructions issue and execute out of order with respect to older non-ready instructions
- Issue queue opens up the pipeline to future independent instructions
 - Tolerate long latencies (cache misses, floating point)
 - Exploit ILP better

Dynamic Scheduling

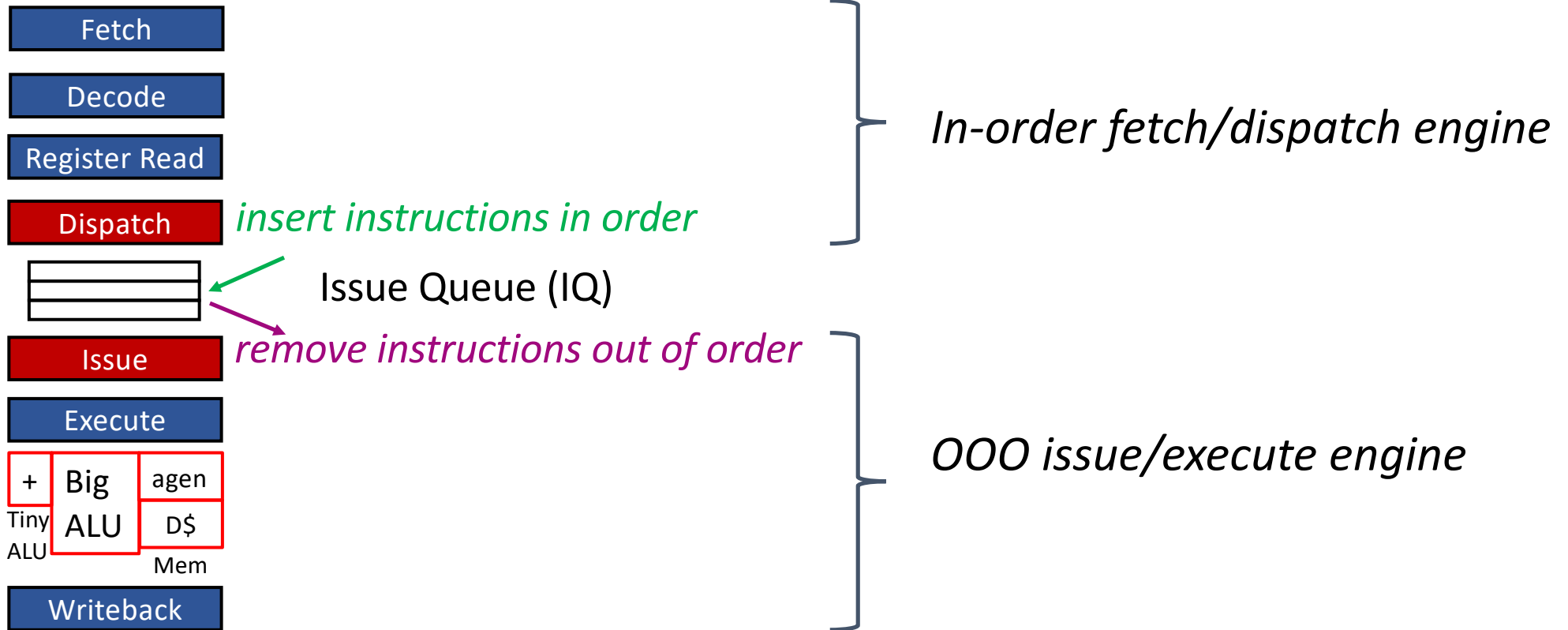
Instruction Scheduling

- **Deciding the order in which instructions are executed, and each ordering is called a schedule**
 - The goal is to maximize ILP
 - All dependences must be respected
- **Static scheduling**
 - Compiler decides the ordering of instruction execution
 - It relies on hardware for correct execution (hazard detection)
 - ARM 5-stage pipeline is statically scheduled
 - 2-way in-order superscalar we saw before is also statically scheduled
- **Dynamic scheduling**
 - Hardware decides the ordering transparently to software
 - “Smart hardware, dumb software”

What is dynamic scheduling?

- **Dynamic scheduling:** Deciding which instructions to execute next, *using extra hardware*, 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**
- Issue queue enables dynamically scheduled processors
- Dynamically scheduled processors are superscalar, **but we will assume scalar execution to simplify illustration of principles**

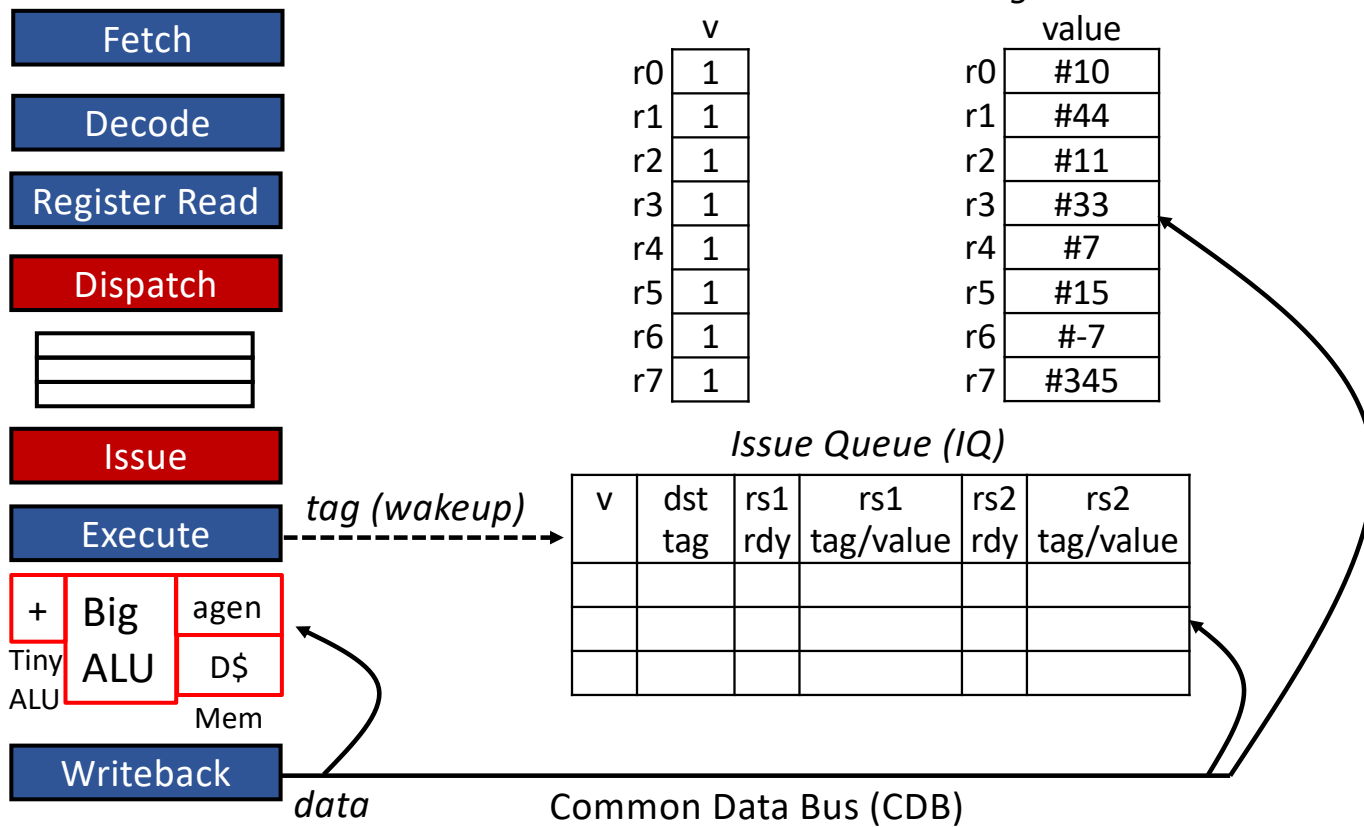
Out-of-Order Scalar Pipeline (v.1)



Version # 1: CDC 6600 Scoreboard

- The idea of a scoreboard was introduced by Control Data Corporation (CDC) in **CDC6600** machine
- **New stages**
 - Dispath stage
 - Issue stage
- **New components**
 - Scoreboard
 - Common Data Bus (CDB)





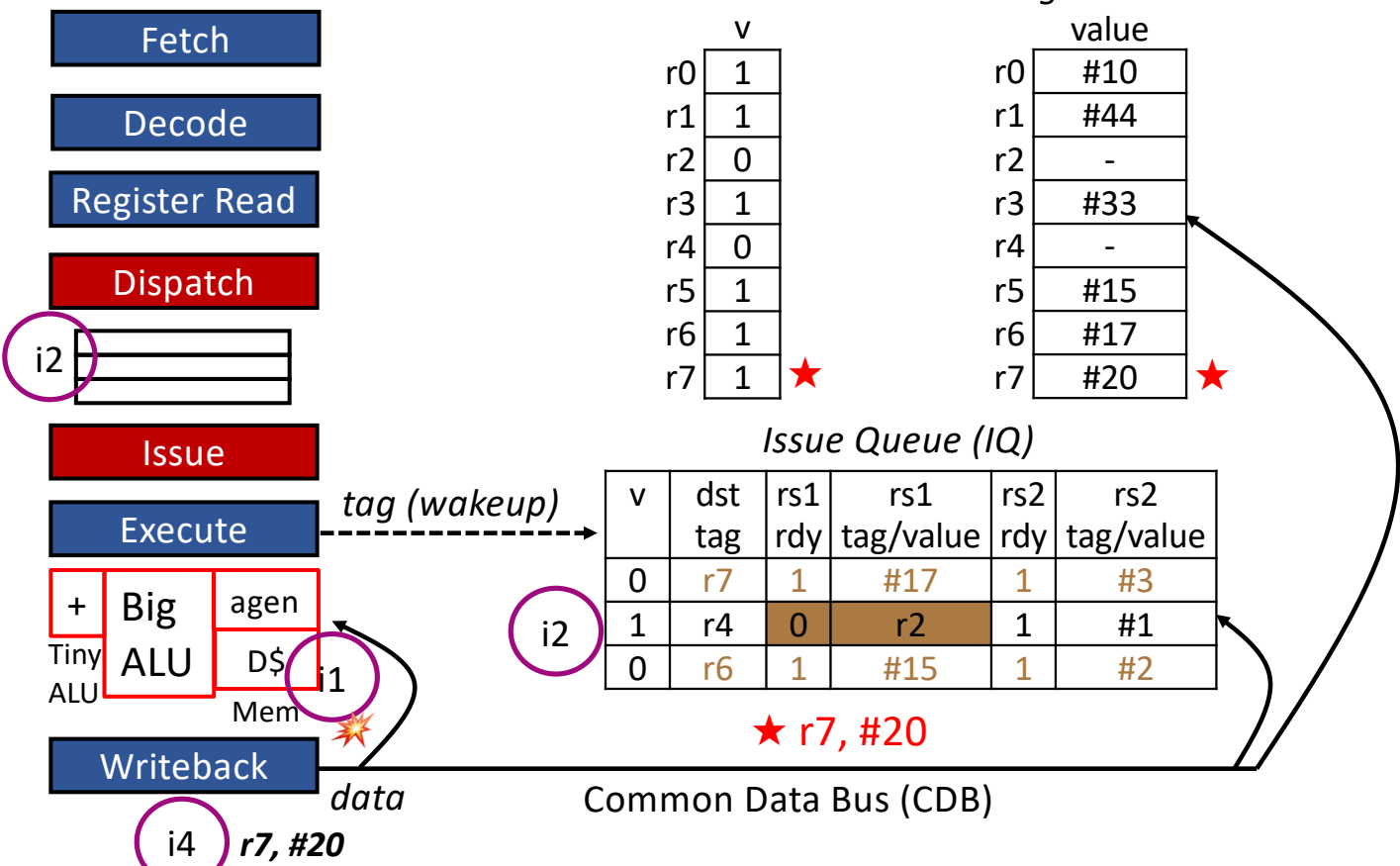
CDC 6600 Scoreboard (Key Additions)

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

CDC 6600 Scoreboard (Key Additions)

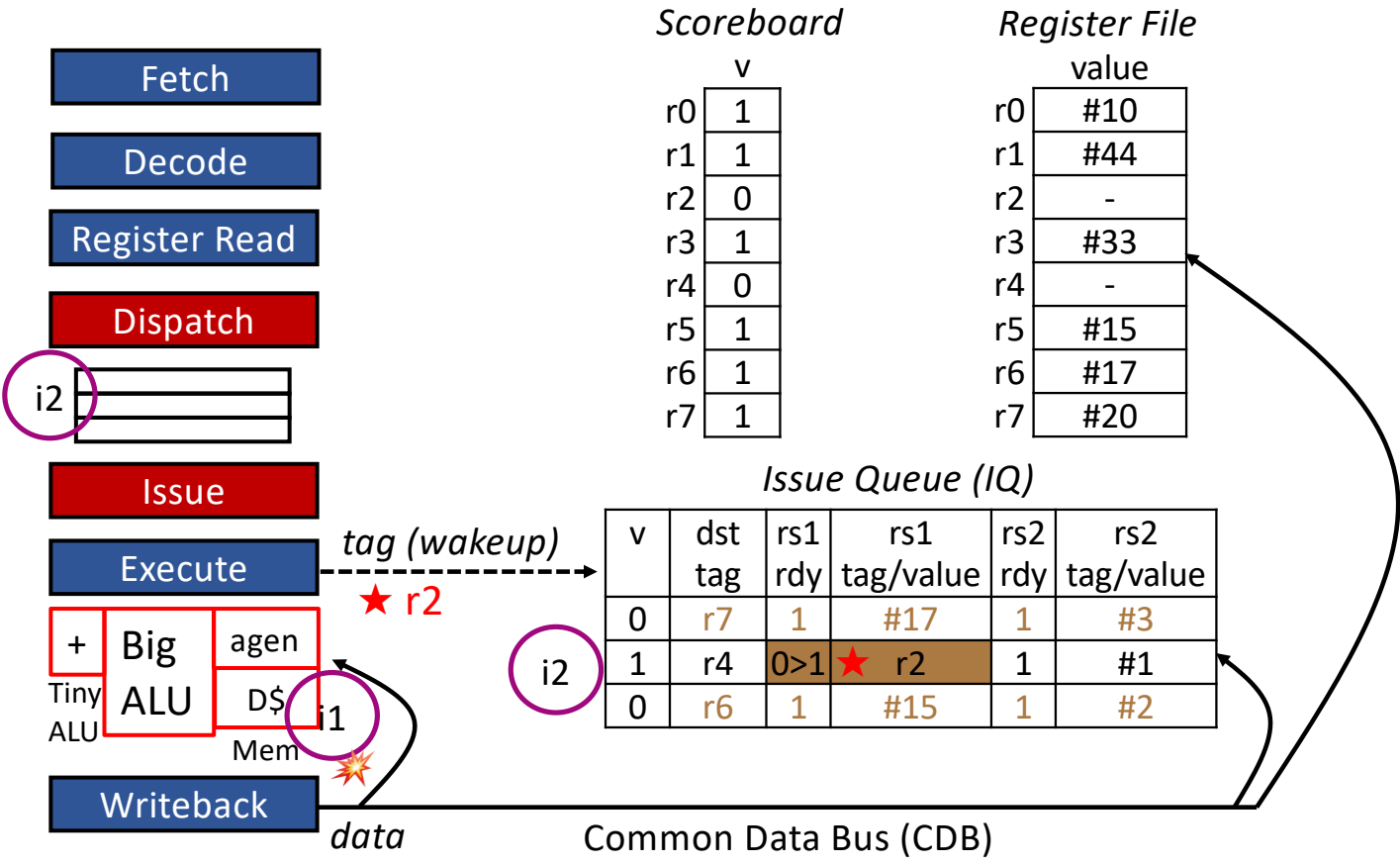
- **How are instructions selected for execution?**
 - The issue queue waits for destination tags to appear
 - When the destination tag appears, it **wakes up** all instructions waiting for that tag
 - One of the “**ready**” instructions is sent to the execute next
 - **Easy way to remember:** Capture-Tag-And-Go-For-Execution

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



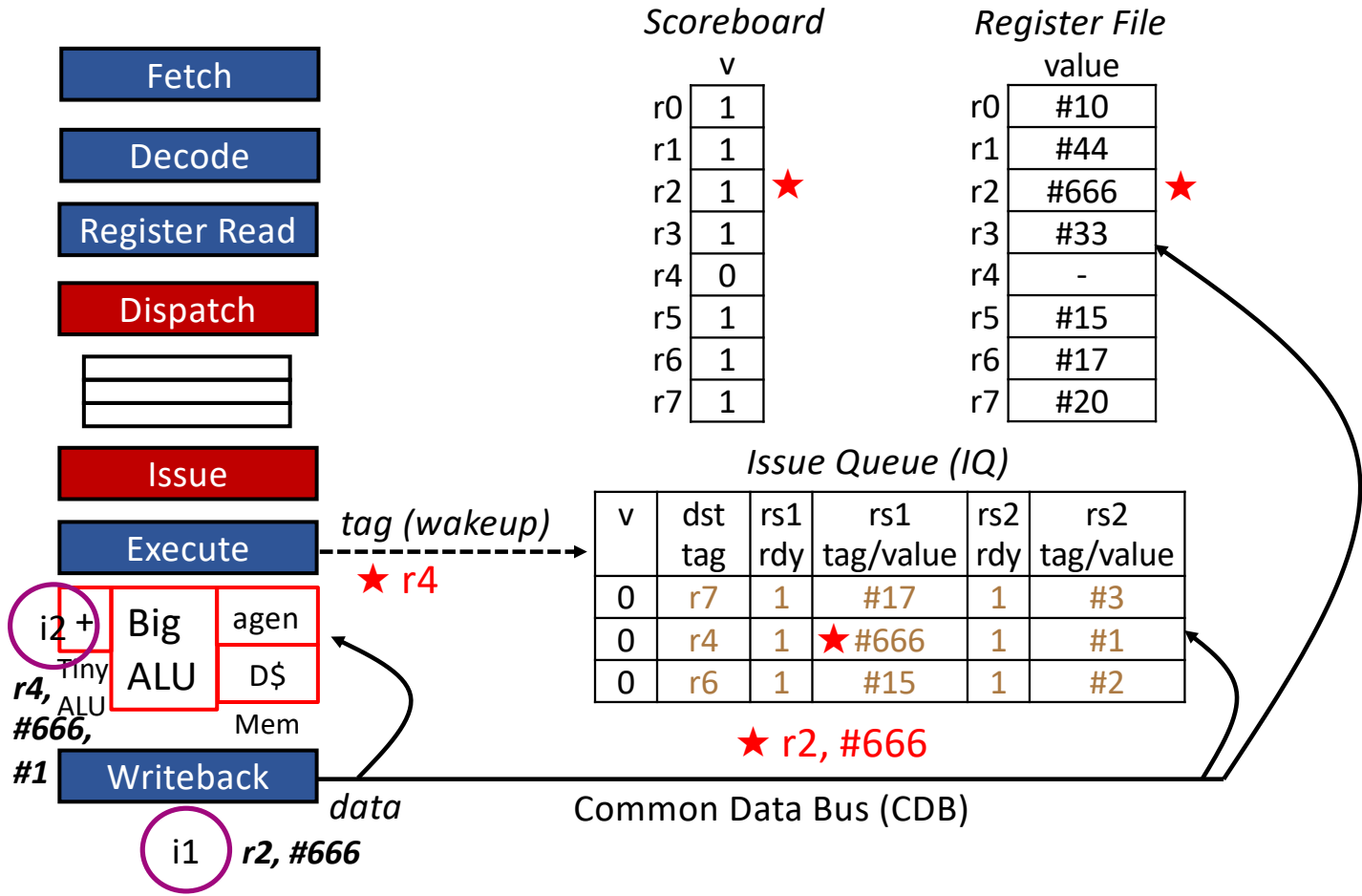
★ cache miss
r2, @44

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

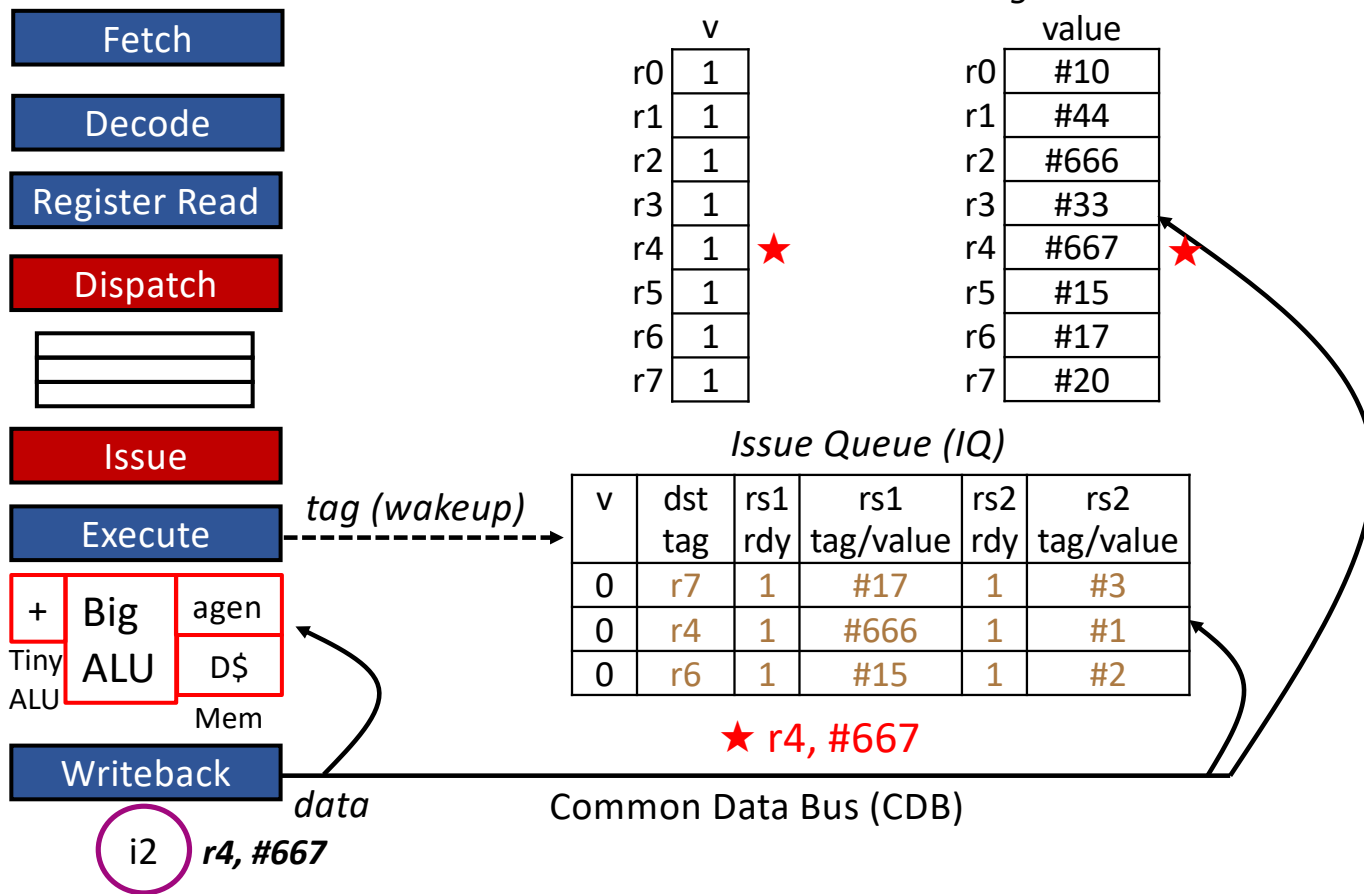


★ cache miss
r2, @44

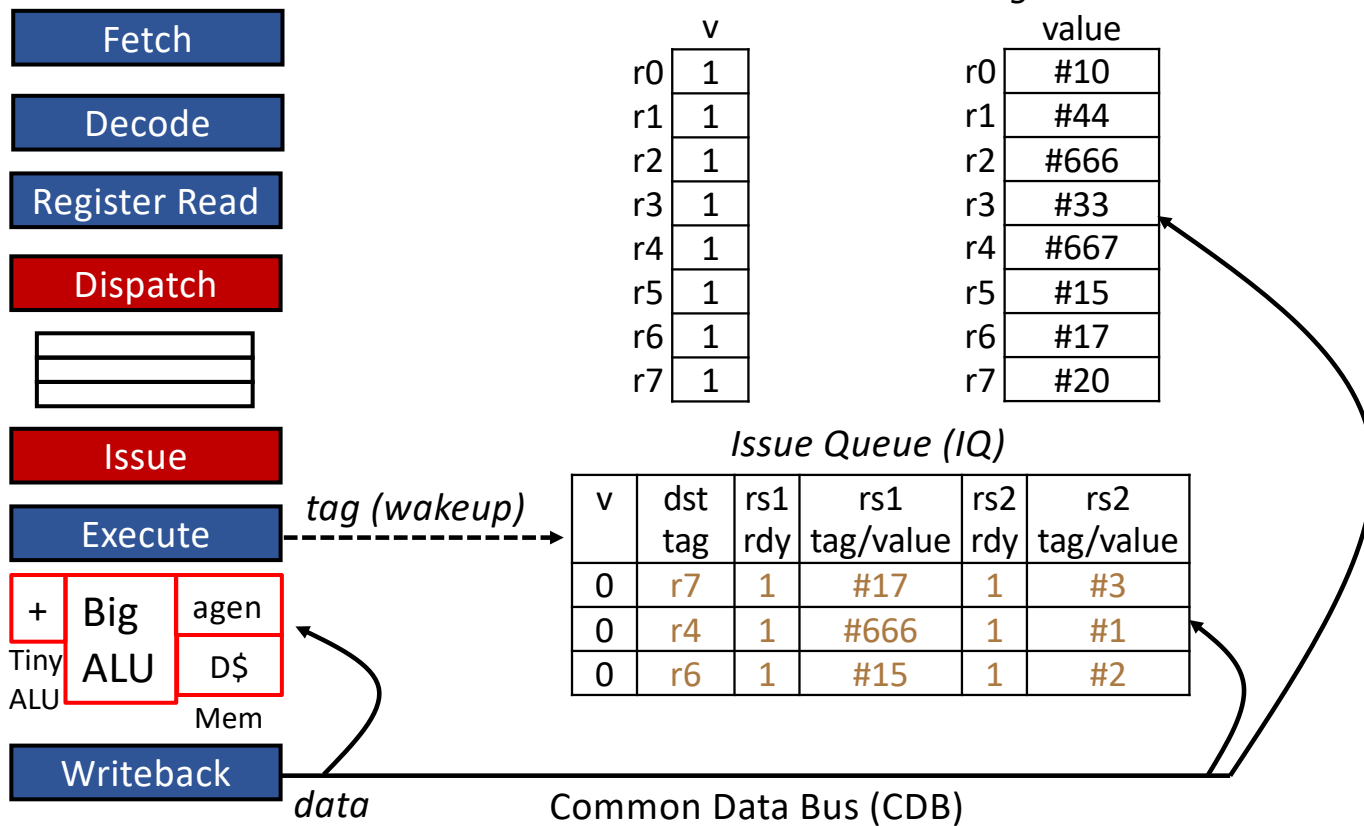
	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
i1: LDR r2, [r1,#0]	FE	DE	RR	DI	IS	EX@	EXD\$... miss ...							
i2: ADD r4, r2, #1		FE	DE	RR	DI	IS	IS	IS	IS	IS							
i3: ADD r6, r5, #2			FE	DE	RR	DI	IS	EX	WB								
i4: ADD r7, r6, #3				FE	DE	RR	DI	IS	EX	WB							



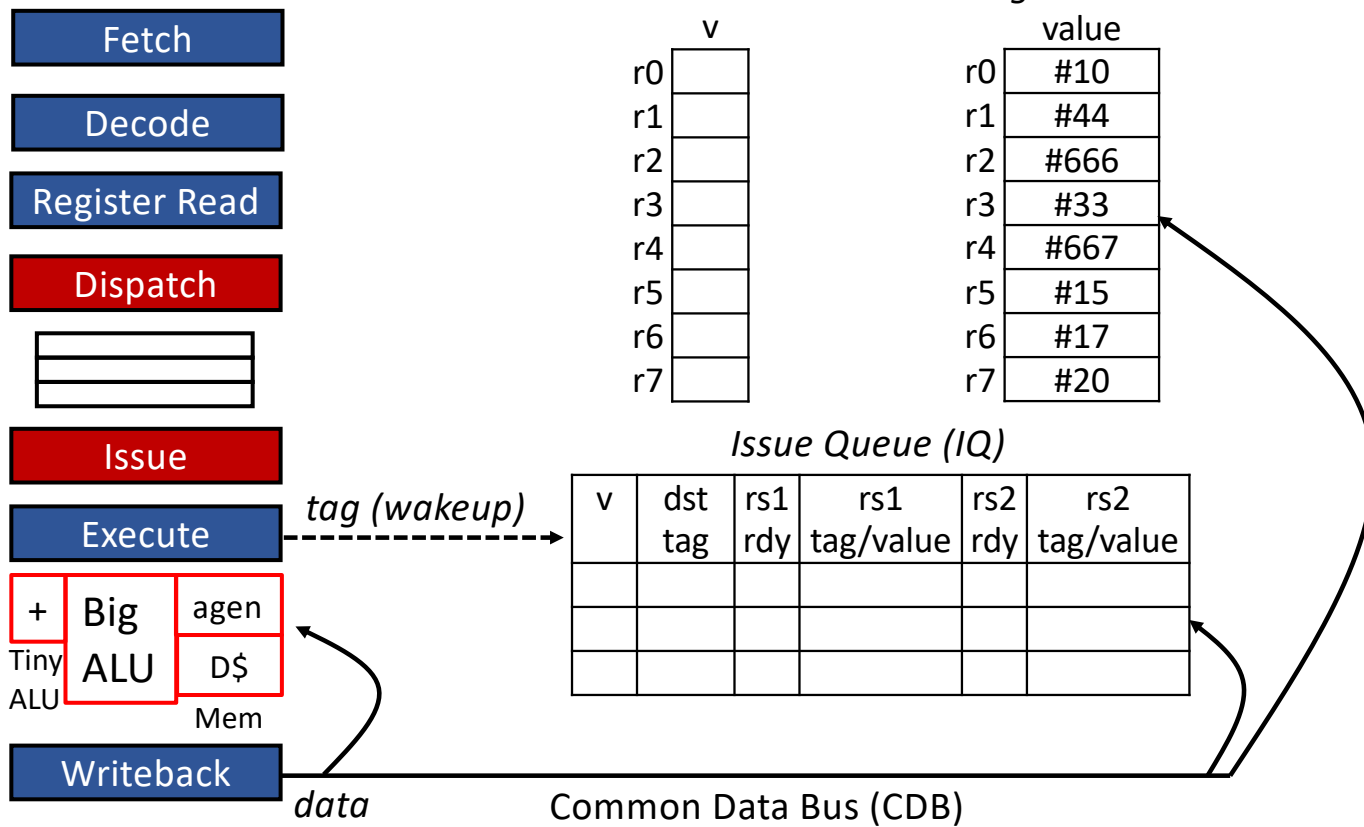
	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
i1: LDR r2, [r1,#0]	FE	DE	RR	DI	IS	EX@	EXD\$...	miss	...	WB					
i2: ADD r4, r2, #1		FE	DE	RR	DI	IS	IS	IS	IS	IS	IS	EX					
i3: ADD r6, r5, #2			FE	DE	RR	DI	IS	EX	WB								
i4: ADD r7, r6, #3				FE	DE	RR	DI	IS	EX	WB							



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



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



	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
i1:	FE	DE	RR	DI	IS	EX _@	EX _{D\$}		... miss ...			WB					
i2:		FE	DE	RR	DI	IS	IS	IS	IS	IS	IS	EX	WB				
i3:			FE	DE	RR	DI	IS	EX	WB								
i4:				FE	DE	RR	DI	IS	EX	WB							

What have we studied?

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

Two problems with OOO v.1

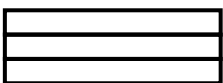
- Cannot recover from misspeculation (aka misprediction)
 - Younger instructions are **speculative** with respect to older instructions
 - Possible to have **older predicted branches** that have not executed yet
- Exceptions are not **precise**, i.e., register file is being updated out of the original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

Fetch

Decode

Register Read

Dispatch



Issue

Execute

i2
#666 != #0
branch to i7
mispredict

Writeback

i1
r2, #666

Scoreboard

v	
r0	1
r1	1
r2	1
r3	1
r4	1
r5	1
r6	1
r7	1

Register File

	value
r0	#10
r1	#44
r2	#666
r3	#33
r4	#7
r5	#15
r6	#17
r7	#20

Can't recover original values of r6, r7

Issue Queue (IQ)

v	dst tag	rs1 rdy	rs1 tag/value	rs2 rdy	rs2 tag/value
0	r7	1	#17	1	#3
0	-	1	#666	1	#0
0	r6	1	#15	1	#2

★ r2, #666

tag (wakeup)

data

Common Data Bus (CDB)

Misprediction detected

Can't flush i3, i4: they are "long gone"

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

Precise Interrupts

Precise Interrupts

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

Example

- In an OOO processor (CDC 6600), the following scenario is perfectly legitimate
 - LDR is waiting for data from memory (cache miss)
 - i3, i4, and i5 have finished execution out of order

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

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

Hardware Speculation

Hardware Speculation

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

Two Humps in a Modern Pipeline

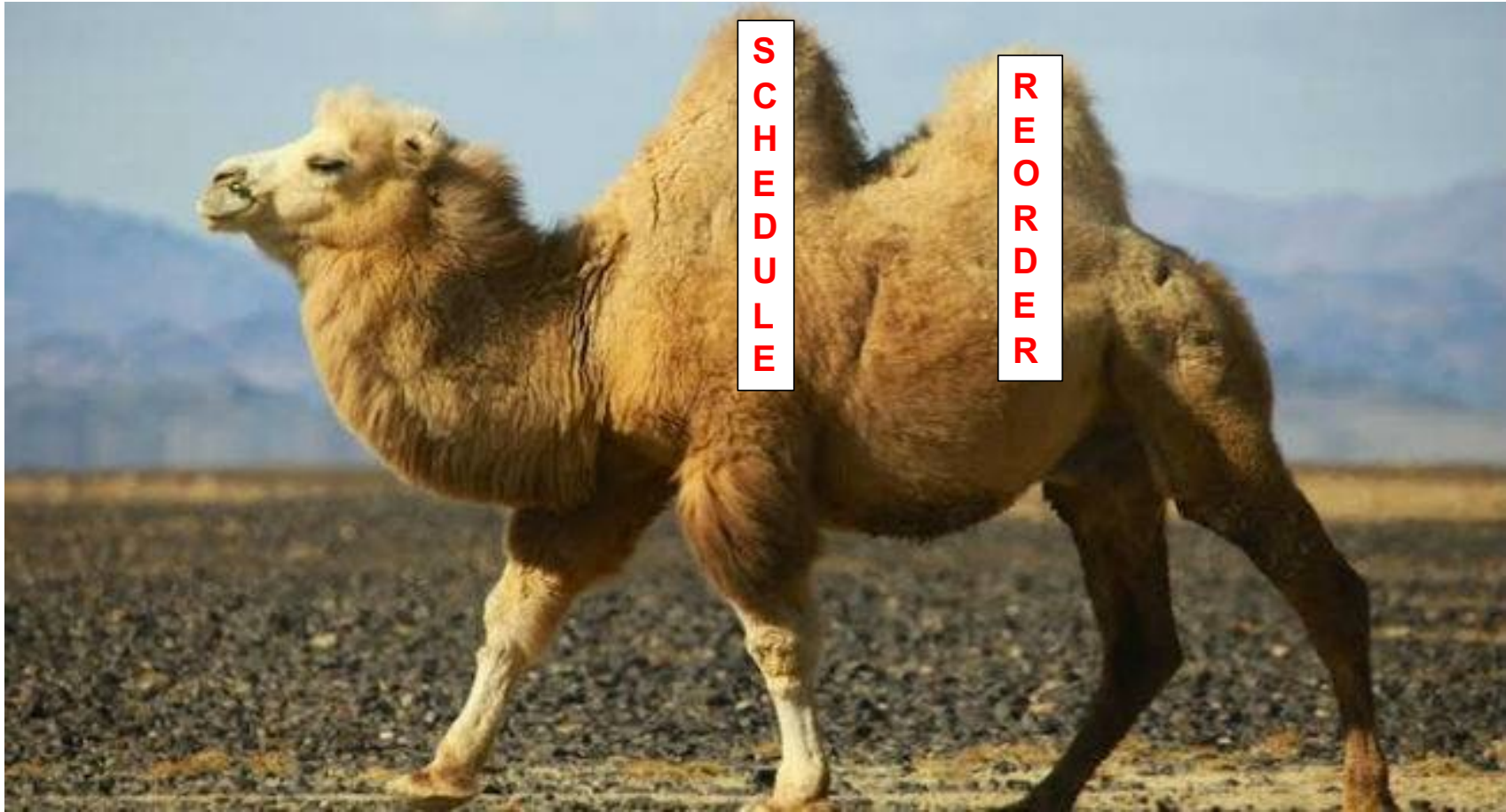


Photo credit: <http://true-wildlife.blogspot.ch/2010/10/bactrian-camel.html>

Out-of-Order Pipeline (v.2)

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

Renaming Example

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

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

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

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

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

Register Renaming: Operational Details

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

Register Renaming

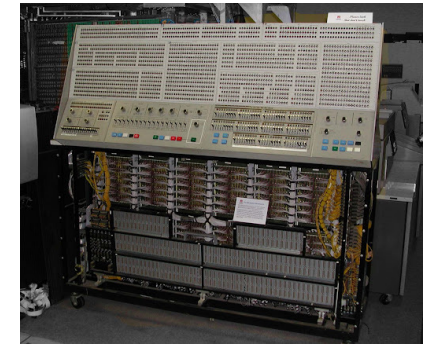
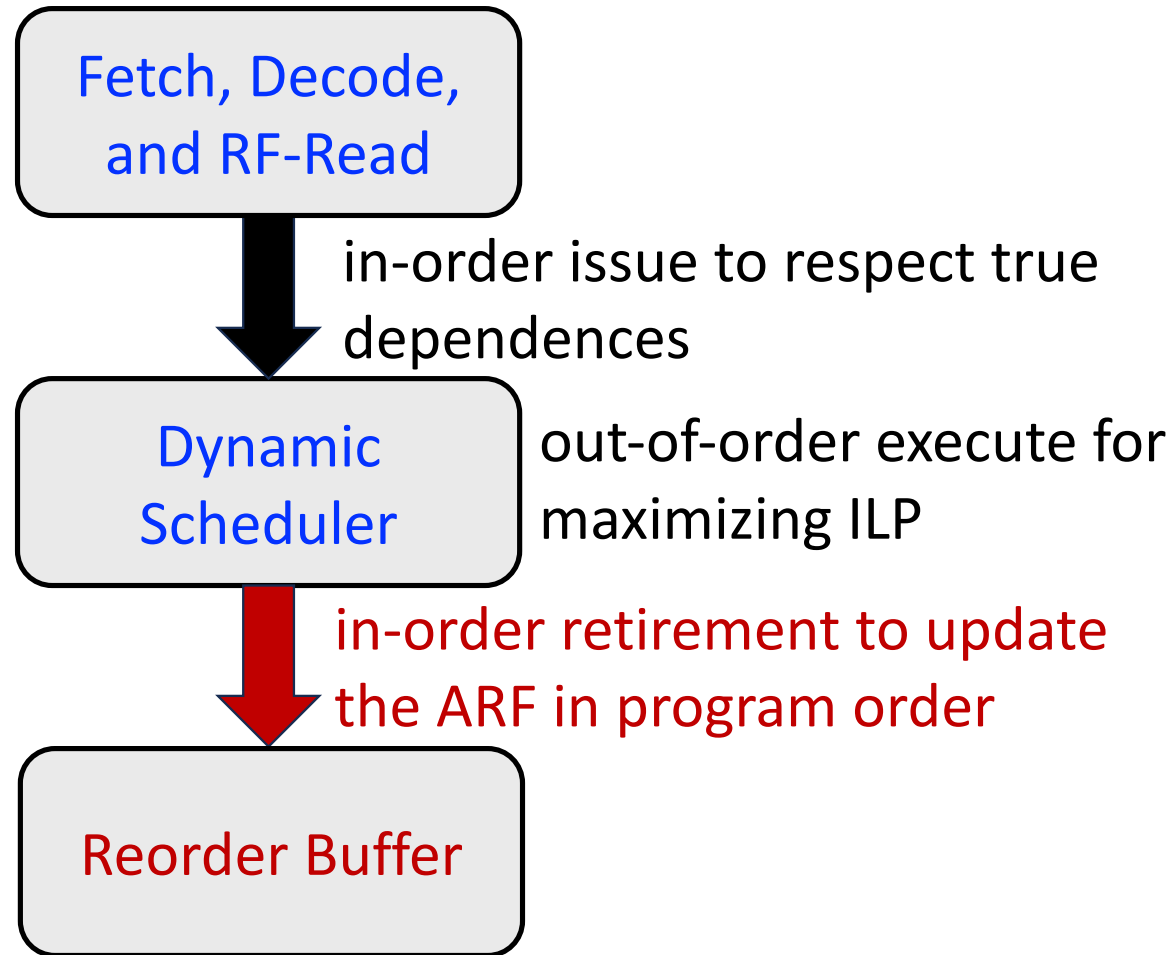
- **Register Read Stage**

- 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

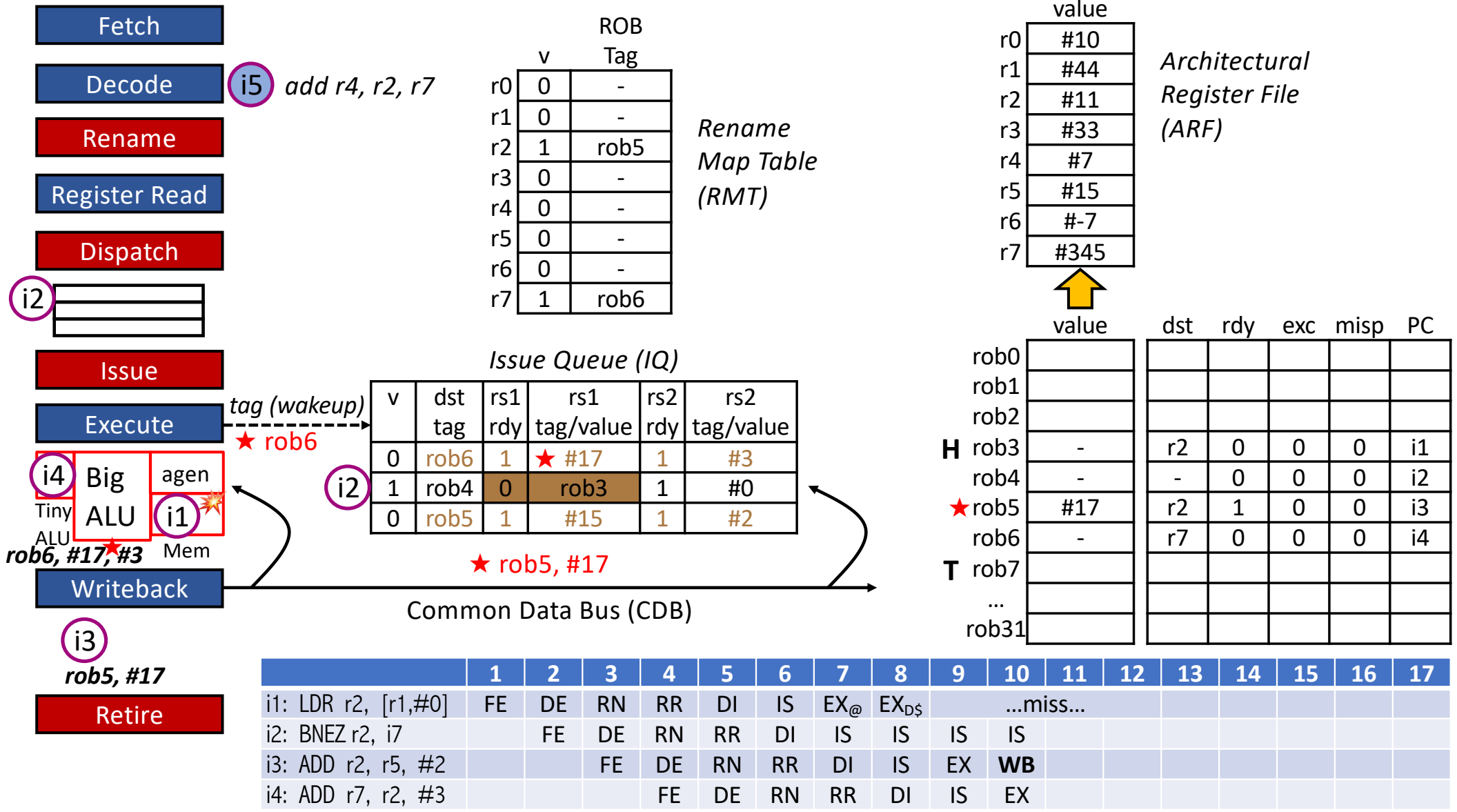
Writeback, Retirement, and Recovery

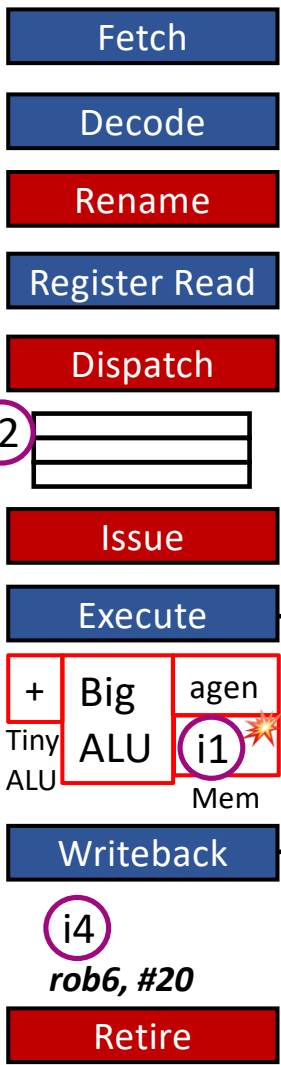
- Writeback Stage
 - Instruction writes its **speculative result** OOO into ROB instead of ARF (at its ROB entry)
- New **Retire** Stage safely **commits** results from **ROB to ARF** in program order
- **Misprediction/exception recovery**
 - Offending instruction posts misprediction or exception bit 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

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



Based on IBM 360/91





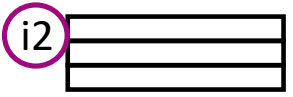
i5 add rob7, rob5, rob6

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

Rename Map Table (RMT)

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

Architectural Register File (ARF)



Issue Queue (IQ)

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

tag (wakeup) ★ rob6

i2

★ rob6, #20

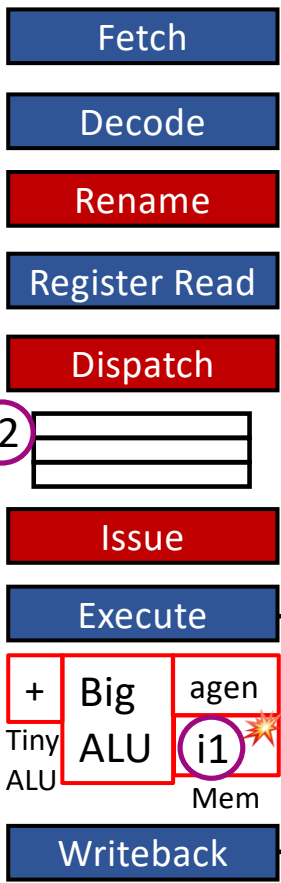
Common Data Bus (CDB)

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

i4
rob6, #20

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

i3



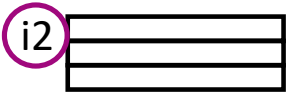
i5: add rob7, #17, #20

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

Rename Map Table (RMT)

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

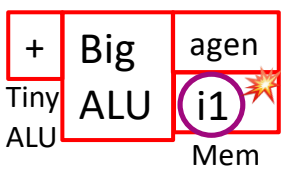
Architectural Register File (ARF)



Issue Queue (IQ)

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

tag (wakeup) ★ rob3



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

Common Data Bus (CDB)

Retire

i3 i4

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

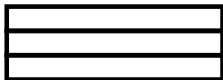
Fetch

Decode

Rename

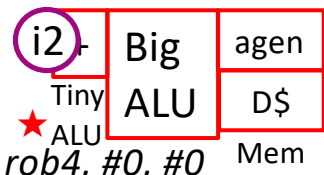
Register Read

Dispatch



Issue

Execute



Writeback

Retire

#0!=#0 is false, not-taken, no misp
rob4, #0, #0

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

Rename Map Table (RMT)

i5 add rob7, #17, #20

Issue Queue (IQ)

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

★ rob3, #0

Common Data Bus (CDB)

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

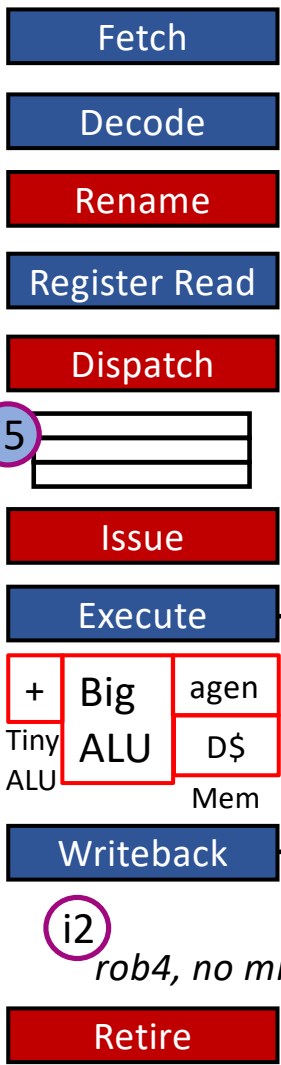
Architectural Register File (ARF)

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

i1 rob3, #0

i3 i4

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



i5

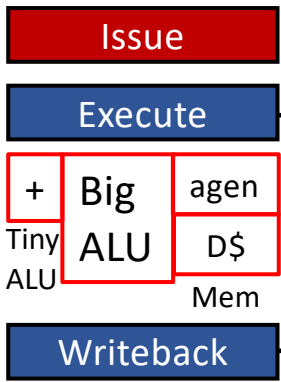
	v	Tag
r0	0	-
r1	0	-
r2	1	rob5
r3	0	-
r4	1	rob7
r5	0	-
r6	0	-
r7	1	rob6

Don't reset:
rob5 != rob3

Rename
Map Table
(RMT)

	value
r0	#10
r1	#44
★ r2	#0
r3	#33
r4	#7
r5	#15
r6	#-7
r7	#345

Architectural
Register File
(ARF)



	v	dst tag	rs1 rdy	rs1 tag/value	rs2 rdy	rs2 tag/value
i5	1	rob7	1	#17	1	#20
	0	rob4	1	#0	1	#0
	0	rob5	1	#15	1	#2

★ rob4, no mispred

Common Data Bus (CDB)

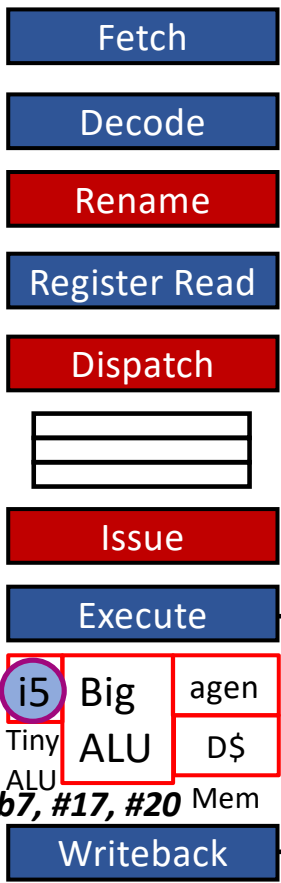
	value	dst	rdy	exc	misp	PC
rob0						
rob1						
rob2						
rob3	#0	r2	1	0	0	i1
★ rob4	-	-	1	0	0	i2
rob5	#17	r2	1	0	0	i3
rob6	#20	r7	1	0	0	i4
rob7	-	r4	0	0	0	i5
T ...						
rob31						

i2
rob4, no misp



i1 i3 i4

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



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

Rename Map Table (RMT)

	value
r0	#10
r1	#44
r2	#0
r3	#33
r4	#7
r5	#15
r6	#-7
r7	#345

Architectural Register File (ARF)

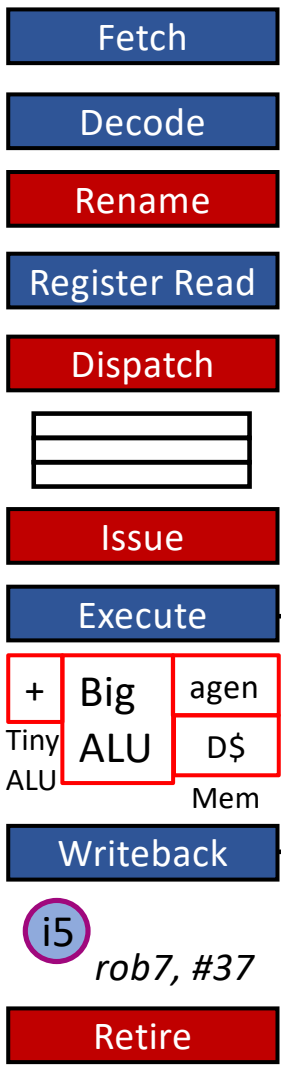
Issue Queue (IQ)					
v	dst tag	rs1 rdy	rs1 tag/value	rs2 rdy	rs2 tag/value
0	rob7	1	#17	1	#20
0	rob4	1	#0	1	#0
0	rob5	1	#15	1	#2

	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	-	r4	0	0	0	i5
...						
rob31						

Common Data Bus (CDB)



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



ROB

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

Reset: rob5 == rob5

Rename Map Table (RMT)

Architectural Register File (ARF)

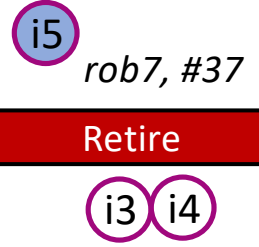
	value
r0	#10
r1	#44
★ r2	#17
r3	#33
r4	#7
r5	#15
r6	#-7
r7	#345

Issue Queue (IQ)

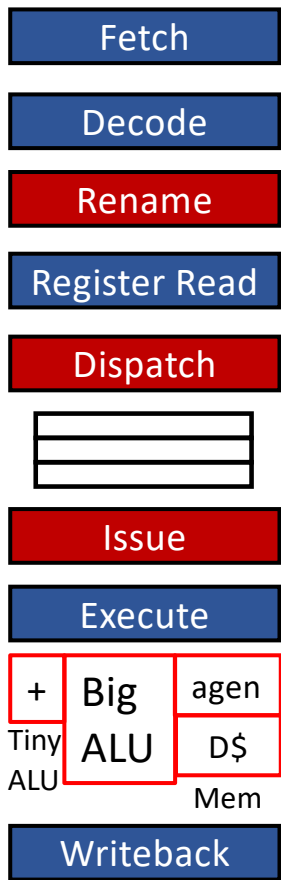
	v	dst tag	rs1 rdy	rs1 tag/value	rs2 rdy	rs2 tag/value
	0	rob7	1	#17	1	#20
	0	rob4	1	#0	1	#0
	0	rob5	1	#15	1	#2

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

★ rob7, #37
Common Data Bus (CDB)



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



ROB

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

Rename Map Table (RMT)

Reset: rob6 == rob6

Architectural Register File (ARF)

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

Issue Queue (IQ)

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

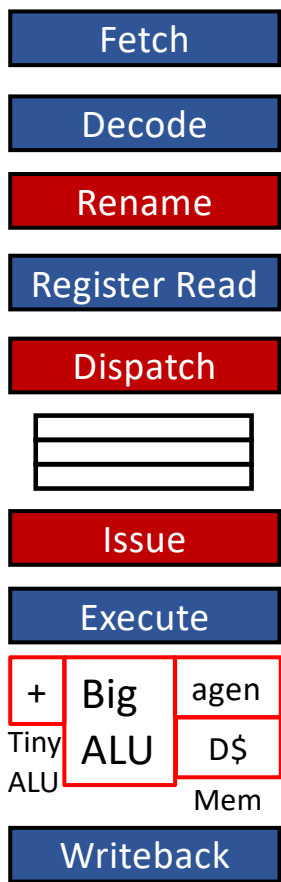
	value	dst	rdy	exc	misp	PC
rob0						
rob1						
rob2						
rob3	#0	r2	1	0	0	i1
rob4	-	-	1	0	0	i2
rob5	#17	r2	1	0	0	i3
rob6	#20	r7	1	0	0	i4
rob7	#37	r4	0	0	0	i5
...						
rob31						

Common Data Bus (CDB)

Retire

i4 i5

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



ROB

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

Rename Map Table (RMT)

Reset: rob7 == rob7

Architectural Register File (ARF)

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

Issue Queue (IQ)

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

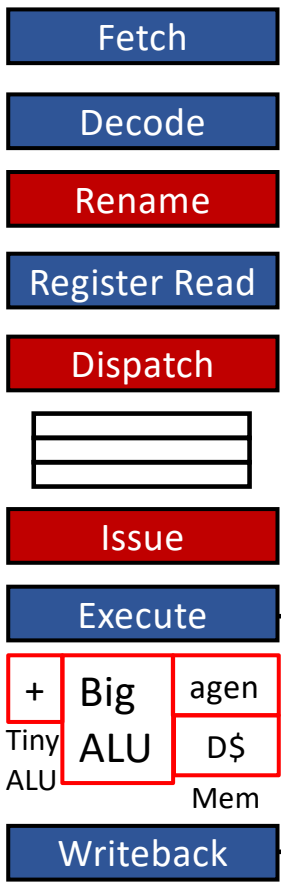
	value	dst	rdy	exc	misp	PC
rob0						
rob1						
rob2						
rob3	#0	r2	1	0	0	i1
rob4	-	-	1	0	0	i2
rob5	#17	r2	1	0	0	i3
rob6	#20	r7	1	0	0	i4
rob7	#37	r4	0	0	0	i5
HT ...						
rob31						

Common Data Bus (CDB)

Retire

i5

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



ROB

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

Rename Map Table (RMT)

value

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

Architectural Register File (ARF)

Issue Queue (IQ)

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

Common Data Bus (CDB)

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

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

Cycle # 1

❖ i1 (Fetch)

Cycle # 2

❖ i1 (Decode)

❖ i2 (Fetch)

Cycle # 3

❖ i_1 (Rename)

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

❖ i_2 (Decode)

❖ i_3 (Fetch)

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

Cycle # 4

- ❖ i_1 (Register Read)
 1. Read the value of the second source operand from the register file: $ARF[r_1]$ is 44
- ❖ i_2 (Rename)
 1. Allocate an entry for i_2 in ROB at rob_4
 2. Rename the destination r_2 to rob_4
 3. Move ROB tail to rob_5
 4. Rename the source operand r_2 to rob_3 because in RMT:
 $v[r_2]=1$
 - ❖ Carry this tag to the issue queue (later) and wait for the value to be produced by the producer (i_1)
- ❖ i_3 (Decode)
- ❖ i_4 (Fetch)

Cycle # 5

- ❖ i_1 (Dispatch)
 1. Instruction is being copied into the issue queue
 - ❖ There are free entries in the issue queue
- ❖ i_2 (Register Read)
 1. Nothing to read from register file (source operand is not ready)
- ❖ i_3 (Rename)
 1. Allocate an entry for i_3 in ROB at rob_5
 2. Rename the destination r_2 to rob_5 , keep $v[r_2]=1$ in RMT
 3. Move ROB tail to rob_6
 4. Rename the source operand r_5 to $ARF[r_5]$ because in RMT:
 $v[r_5]=0$
- ❖ i_4 (Decode)

Cycle # 6

- ❖ i_1 (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 (rs_1 $rdy=1$ and rs_2 $rdy=1$)
- ❖ i_2 (Dispatch)
 1. Instruction is being copied into the issue queue
- ❖ i_3 (Register Read)
 1. Read $ARF[r_5]=\#15$
- ❖ i_4 (Rename)
 1. Allocate an entry for i_4 in ROB at rob_6 (tail moves to r_7)
 2. Rename the destination r_7 to rob_6 , set $v[r_7]=1$ in RMT
 3. Rename r_2 to rob_5 because in RMT: $v[r_2]=1$

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!)

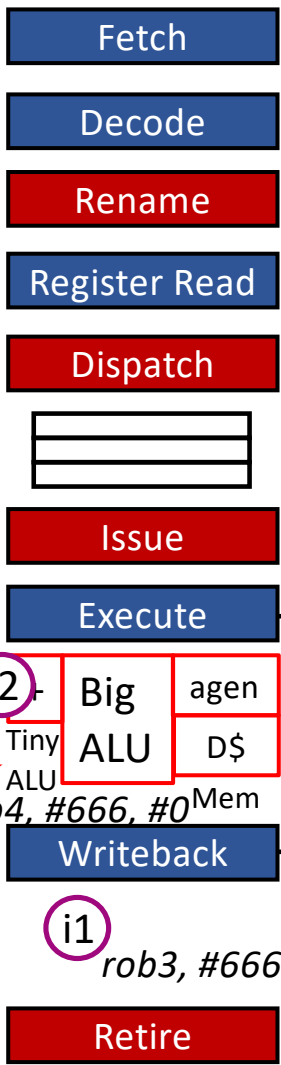
Observations

- **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



#666!=#0 is true, taken, misp
rob4, #0, #0

i5 add rob7, #17, #20

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

Rename Map Table (RMT)

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

Architectural Register File (ARF)

Issue Queue (IQ)

	v	dst tag	rs1 rdy	rs1 tag/value	rs2 rdy	rs2 tag/value
	0	rob6	1	#17	1	#3
	0	rob4	1	#666 ★	1	#0
	0	rob5	1	#15	1	#2

★ rob3, #666

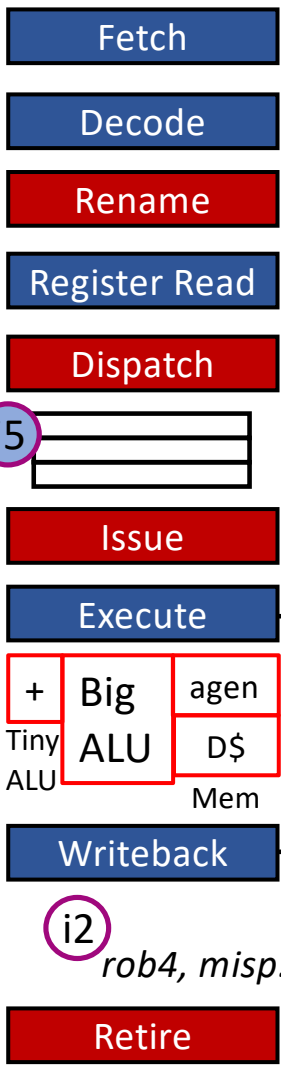
Common Data Bus (CDB)

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

i1 rob3, #666

i3 i4

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



i5

i2
rob4, misp.

i1 i3 i4

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

Don't reset: rob5 != rob3

Rename Map Table (RMT)

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

Architectural Register File (ARF)

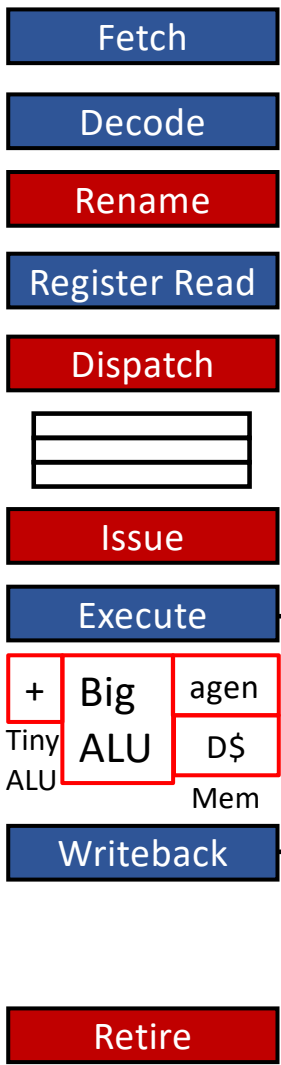
v	dst tag	rs1 rdy	rs1 tag/value	rs2 rdy	rs2 tag/value
i5	rob7	1	#17	1	#20
0	rob4	1	#666	1	#0
0	rob5	1	#15	1	#2

★ rob4, misp.

Common Data Bus (CDB)

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

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
i1: LDR r2, [r1,#0]	FE	DE	RN	RR	DI	IS	EX@	EXD\$...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			



★ Recover RMT by flash-cleaning all valid bits

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

Rename Map Table (RMT)

★ Flush ROB by setting T=H

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

Architectural Register File (ARF)

Issue Queue (IQ)

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

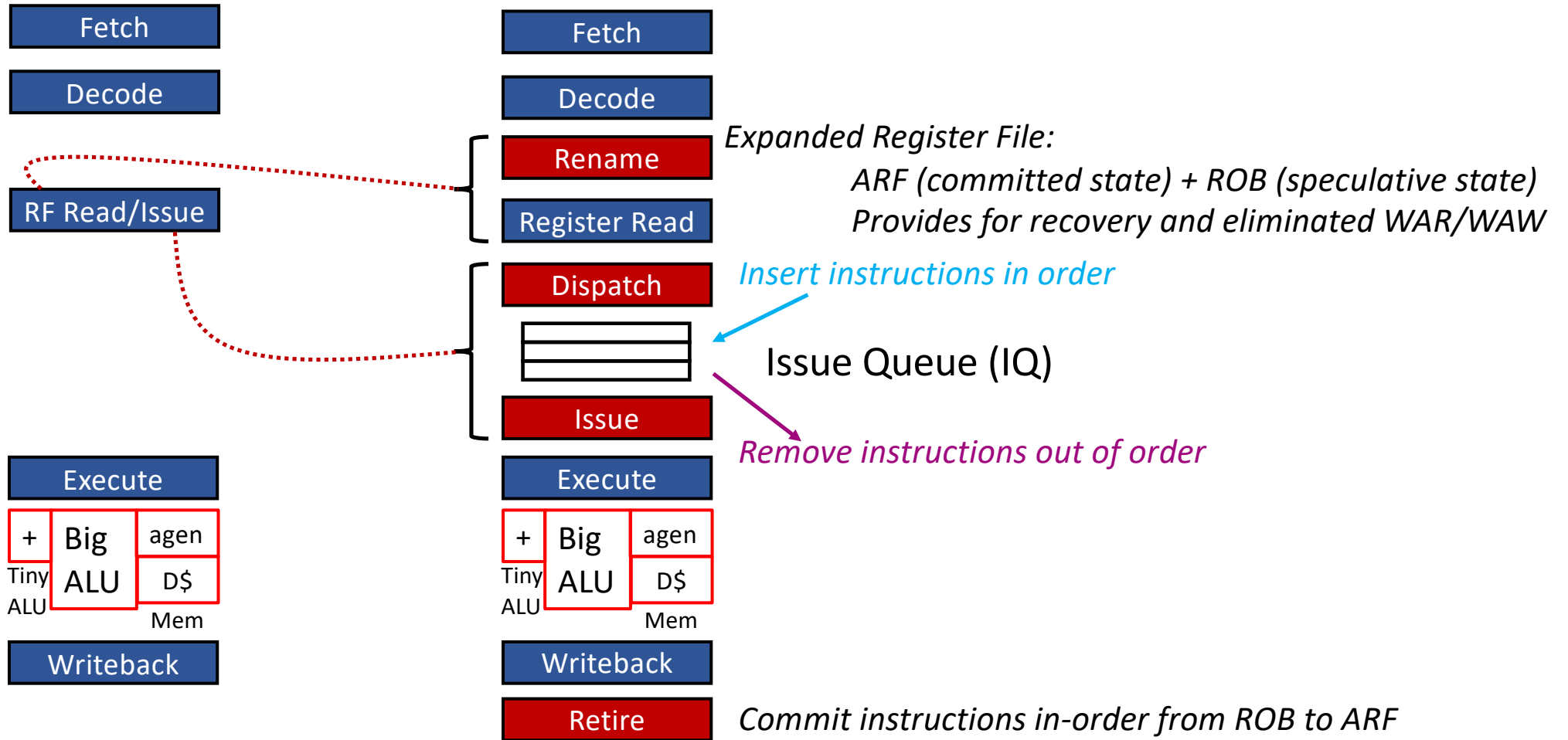
★ Flush all pipeline stages. (i5) They contain younger instructions than the ROB head, i.e., branch

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

i2

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
i1: LDR r2, [r1,#0]	FE	DE	RN	RR	DI	IS	EX@	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



OOO Execution of Loads and Stores

- Loads and stores also execute out of order
- A load and a store can be done safely out of order, provided they access different addresses
- If a load and store access the same address, then the reordering could result in a hazard
 - Load is before the store in program order (WAR)
 - Store is before the load in program order (RAW)

Speculative Load Execution

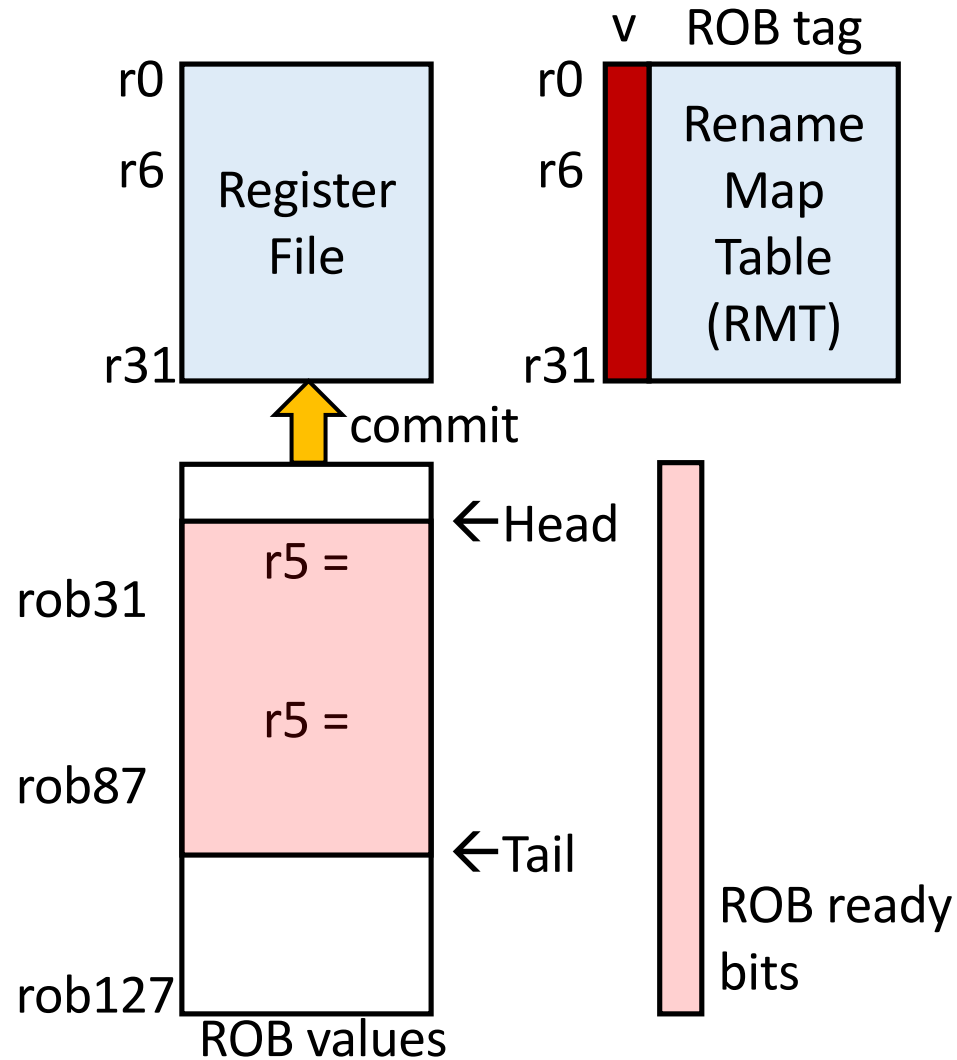
- Load execute speculatively, i.e., before the addresses of all older stores are known
- Load searches for all speculative (older) stores with matching addresses
 - Load gets the “best it can get” (cache, main memory, ROB, RF,)
 - If it gets a stale value, recovery mechanism will save the day
- On execution,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

IBM 360/91 Floating Point Unit



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

ARF + ROB Summary



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

Recovery

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

Revision: Main Concepts

- **Dynamic branch prediction**
 - Choosing which instructions to execute next
- **Speculation**
 - Allowing the execution of instructions before control dependences are resolved (with the ability to undo the effects of an incorrectly **speculated** sequence)
- **Dynamic scheduling**
 - Dealing with the scheduling of different combinations of instructions
 - Dynamic scheduling with speculation exploits limited ILP as branches must resolve prior to actually executing instructions in the predicted path
- **Register renaming**
 - Renaming logical registers to an extended set of physical registers to avoid WAR/WAW
- **Hardware-based speculation**
 - Dynamic branch prediction + dynamic scheduling + speculation
 - Renaming is an **optional** but **important** optimization
- **Precise interrupts**
 - On an exception, the architectural state must correspond to the sequential execution

Modern Processor Design: Key to High Performance

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

Modern Processor Design: Key to High Performance

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

A Historical Debate

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

ILP Exploitation: Three Classes

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