COMP3710 (Class # 5176)
Special Topics in Computer Science
Computer Microarchitecture

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

Australian National University

Plan

Week 5: In-order to out-of-order (OOO) transformation

Week 6: OOO v.1 (CDC 6600 Scoreboard) and OOO v.2 (IBM 360/91)

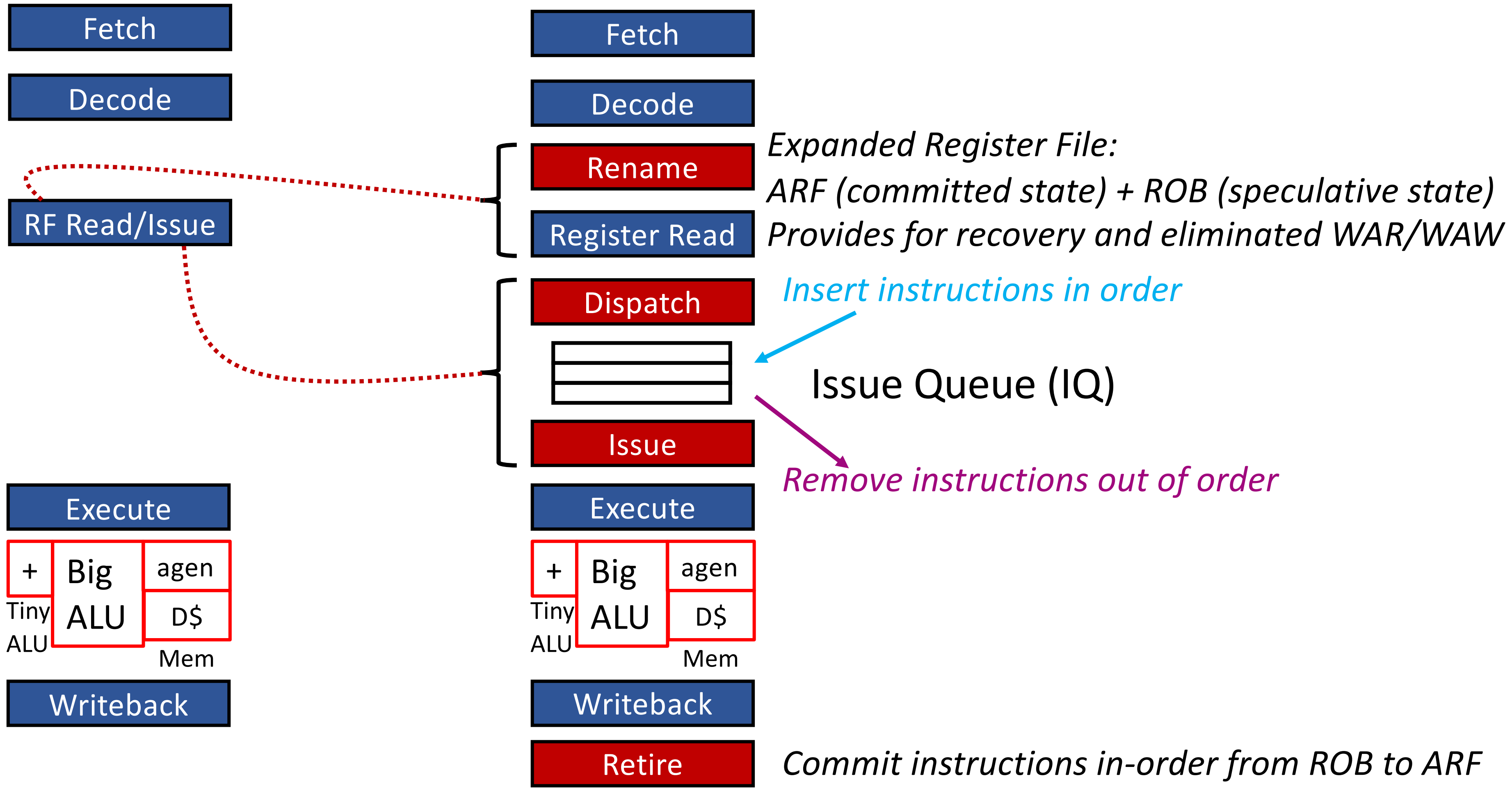
Week 7: OOO v.3 a.k.a. Physical Register File (PRF) microarchitecture

Week 7: Load/Store queue and the load/store execution lane

Week 8+9: Cache design/implementation (assignment # 2)

Remaining topics: ★Virtual memory★, SMT, Multicores, DRAM, NVM

In-order to Out-of-Order



Expanded Register File:

ARF (committed state) + ROB (speculative state)

Provides for recovery and eliminated WAR/WAW

Insert instructions in order

Remove instructions out of order

Commit instructions in-order from ROB to ARF

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)

Some Notes

Precise interrupts

- Must deal with this problem with or without speculation
- Dynamic scheduling alone makes precise interrupts challenging
- IBM 360/91 had dynamic scheduling but no precise interrupts (no ROB)

Speculation

- Recovery and repair mechanisms are a must have
- Cannot have branch prediction but no recovery
- IBM 360/91 had no speculation (i.e., no ROB and stall-on-branch)

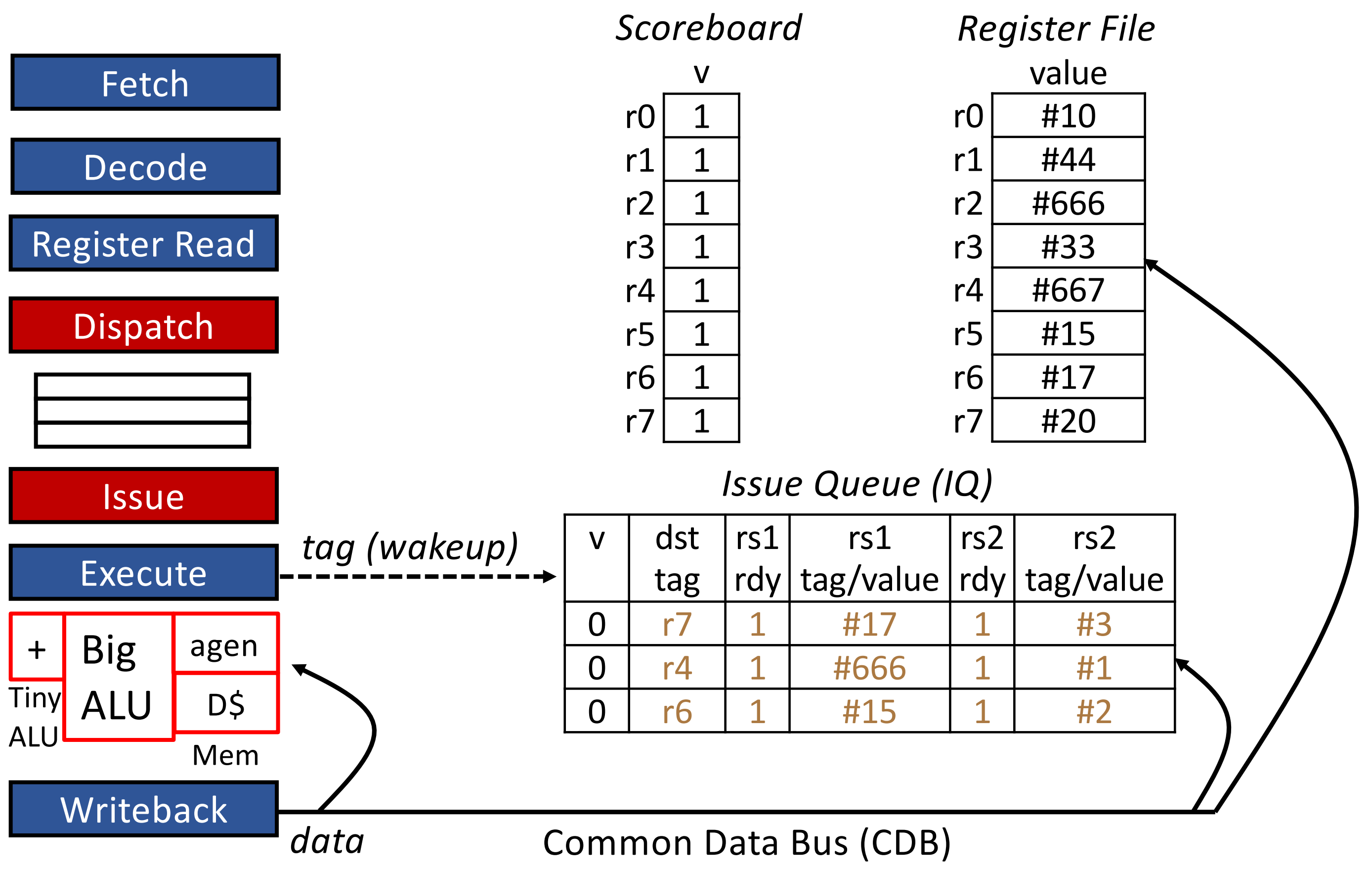
Hardware speculation (dynamic scheduling + renaming + speculation)

- Dynamic scheduling + speculation: a very happy marriage
- In contrast, dynamic scheduling alone only partially overlaps instructions (limited ILP)
- Without speculation, OOO cannot schedule past a single basic block
  - Basic block: Straight-line piece of code with no branches in except at the entry and no branches out except at the exit

Quiz

Which of the following features were present in CDC 6600 scoreboard?

- Dynamic scheduling
- Register renaming
- Speculation
- Precise interrupts
- Hardware speculation



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	#667
r5	#15
r6	#17
r7	#20

Issue Queue (IQ)

v	dst tag	rs1 rdy	rs1 tag/value	rs2 rdy	rs2 tag/value
0	r7	1	#17	1	#3
0	r4	1	#666	1	#1
0	r6	1	#15	1	#2

	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							

Quiz

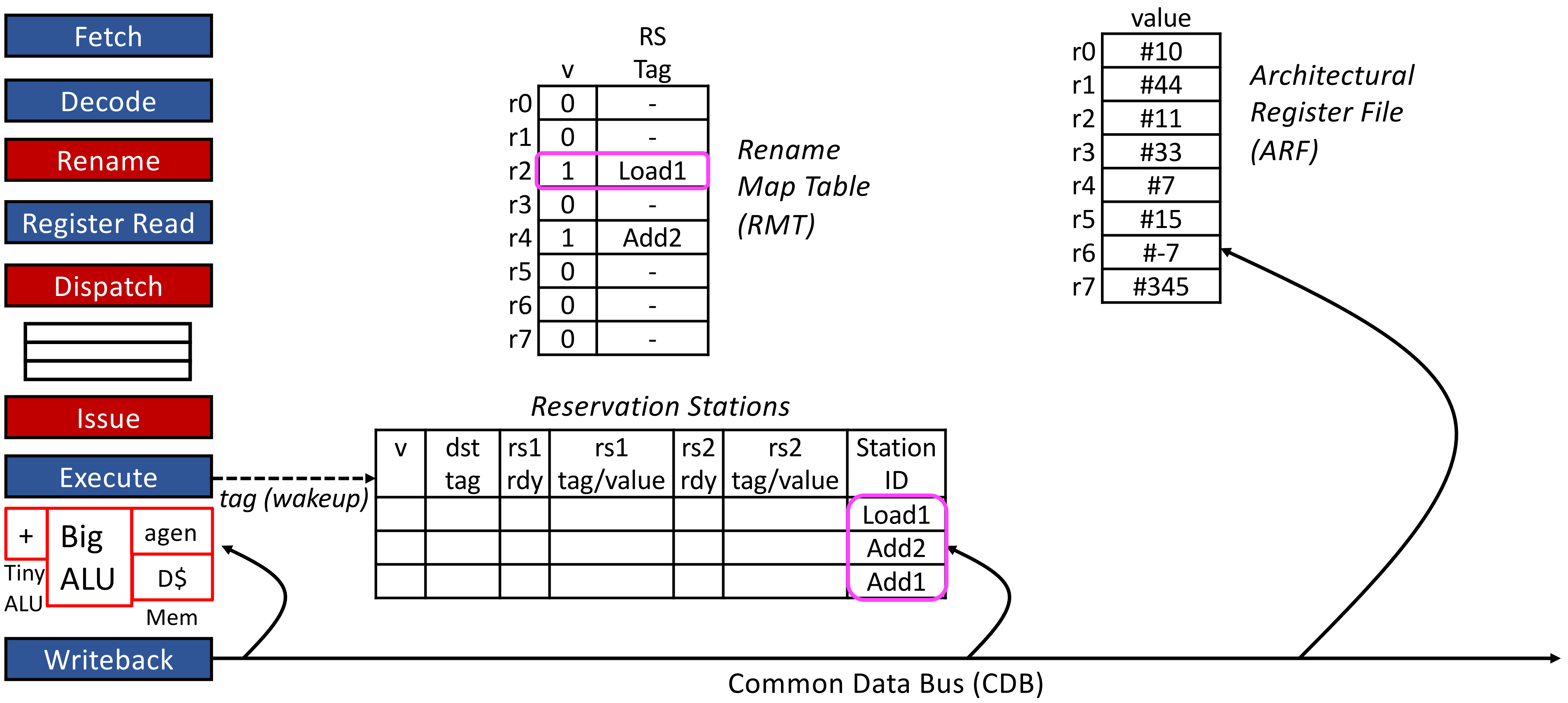
Which of the following features were present in CDC 6600 scoreboard?

- Dynamic scheduling
- Register renaming
- Speculation
- Precise interrupts
- Hardware speculation

Quiz

Which of the following features were present in the initial IBM 360/91?

- Dynamic scheduling
- Register renaming
- Speculation
- Precise interrupts
- Hardware speculation



Rename Map Table (RMT)

	v	RS Tag
r0	0	-
r1	0	-
r2	1	Load1
r3	0	-
r4	1	Add2
r5	0	-
r6	0	-
r7	0	-

Architectural Register File (ARF)

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

Reservation Stations

v	dst tag	rs1 rdy	rs1 tag/value	rs2 rdy	rs2 tag/value	Station ID
						Load1
						Add2
						Add1

Removing ROB to Mimic 360/91

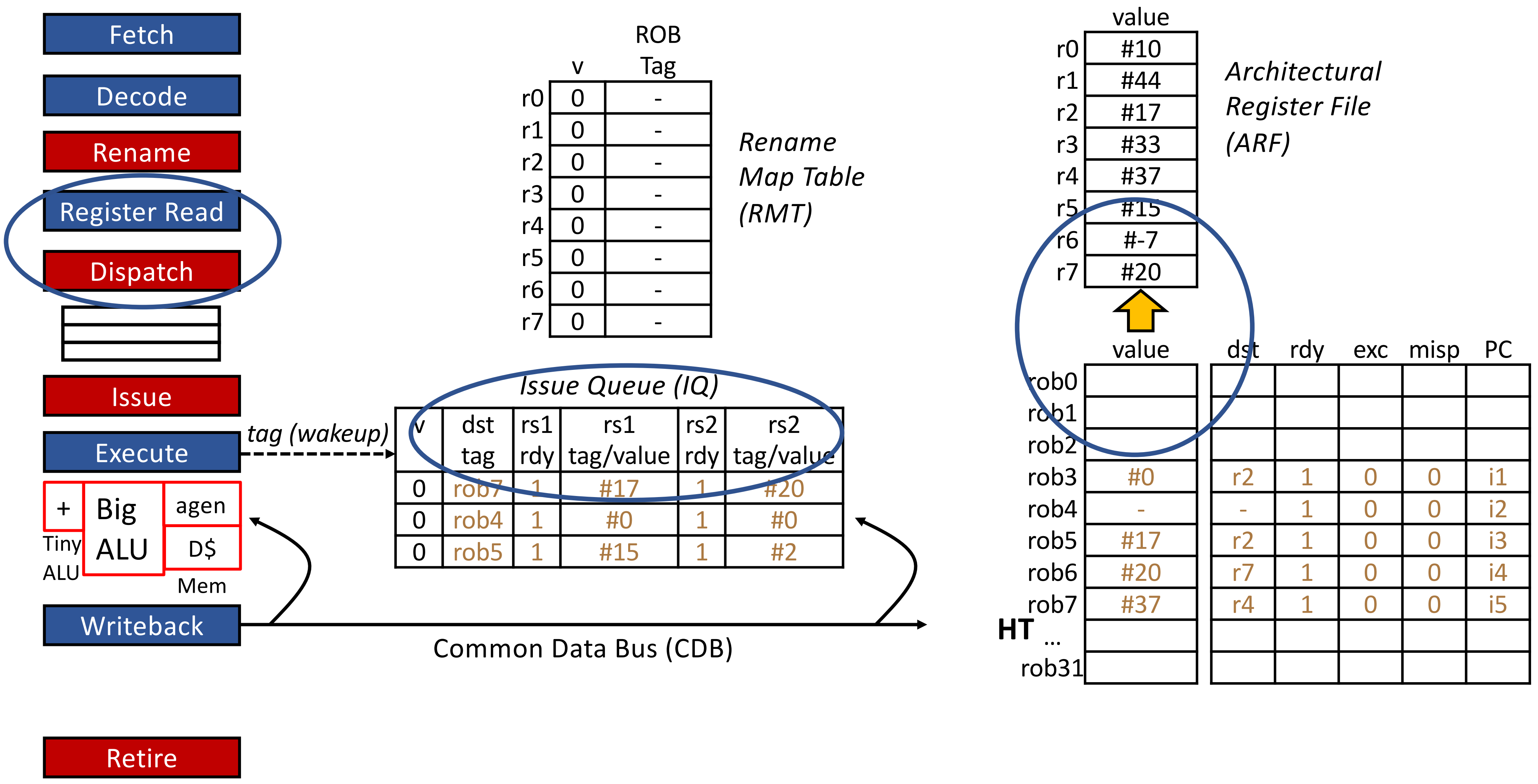
Quiz

Which of the following features were present in the initial IBM 360/91?

- Dynamic scheduling
- Register renaming
- Speculation
- Precise interrupts
- Hardware speculation

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



Rename Map Table (RMT)

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

Architectural Register File (ARF)

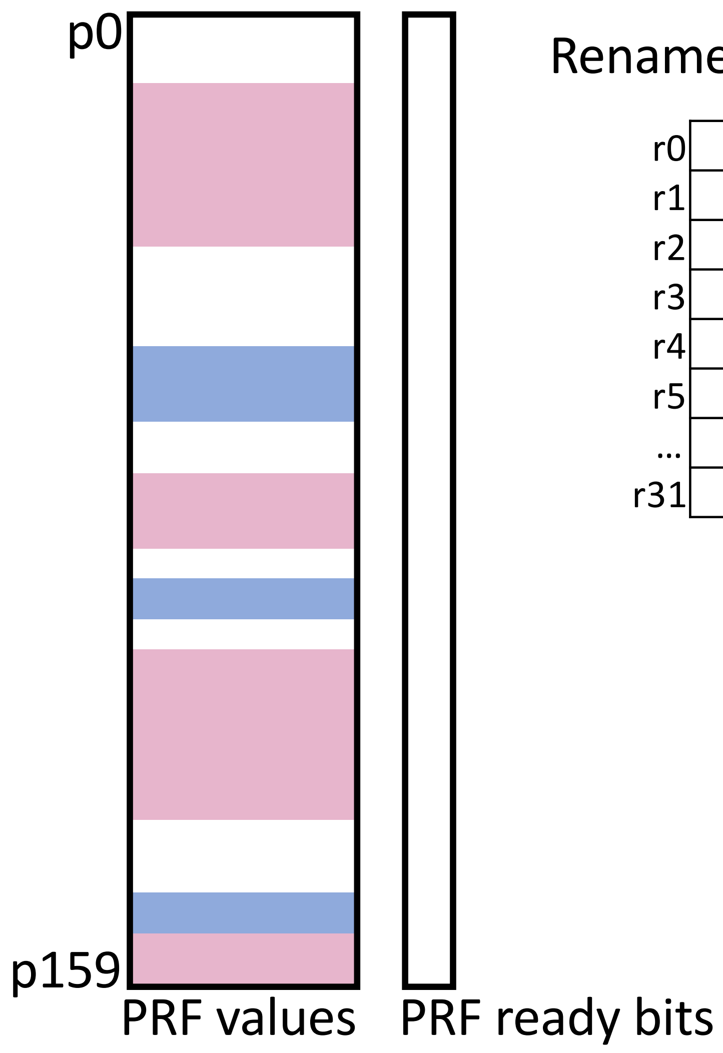
	value
r0	#10
r1	#44
r2	#17
r3	#33
r4	#37
r5	#15
r6	#-7
r7	#20

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	1	0	0	i5
…						
rob31						

PRF Style



Rename Map Table

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

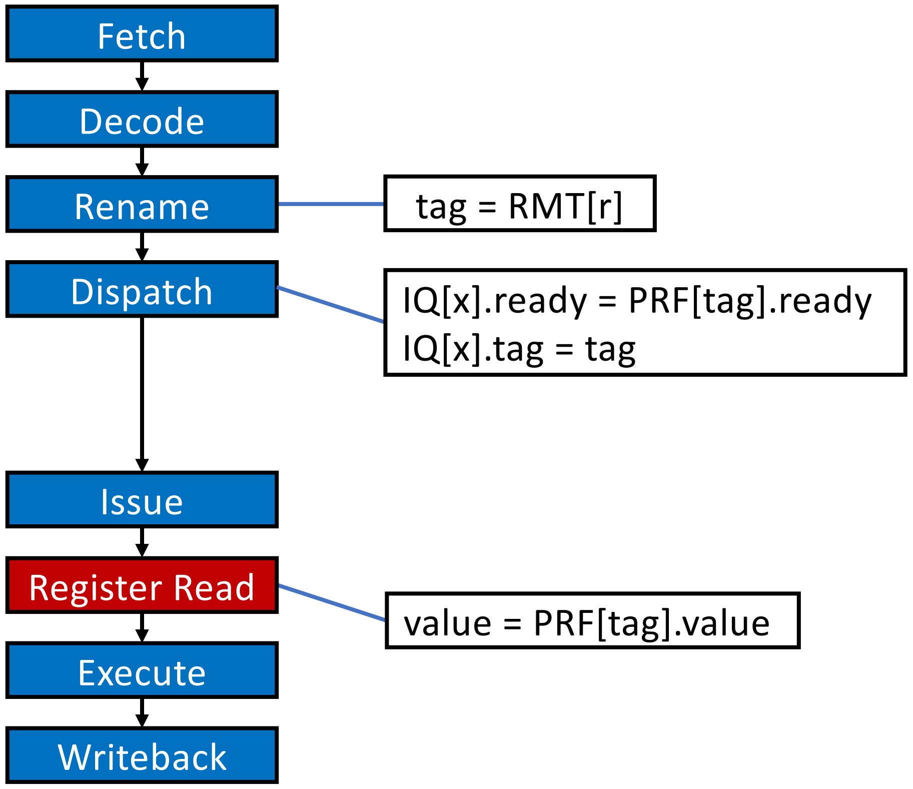
Compared to ARF + ROB

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

PRF Style Pipeline

Compared to ARF + ROB

- "Register Read" stage after "issue" stage
- Issue Queue (IQ) contains no values, just tags



Logical and Physical Registers

Logical registers

- Also called program (architectural) registers
- MIPS : 32
- Alpha: 32
- 8080: 8
- x86-64: 16 (without AVX)

Physical registers

- Also called hardware (microarchitectural) registers
- Intel Sandybridge: 160
- Apple A14 (Firestorm): 300+

https://www.anandtech.com/show/16226/apple-silicon-m1-a14-deep-dive/2
https://www.anandtech.com/show/3922/intels-sandy-bridge-architecture-exposed/3

Renaming

- The renaming problem
  - Assign a unique physical register to a logical register
  - Assign multiple definitions of a logical register a unique physical register

- Structures for renaming
  - Physical Register File (PRF)
  - Rename Map Table (RMT)
  - Free list (list of unused registers in PRF)
    - We will see in a few slides how free list is built
    - First, let’s understand the renaming process assuming RMT and free list are in a specific state

Renaming Example

```
load  r1,  16(r2)
add   r3,  r1,  #1
load  r1,  20(r2)
sub   r4,  r1,  #1
```

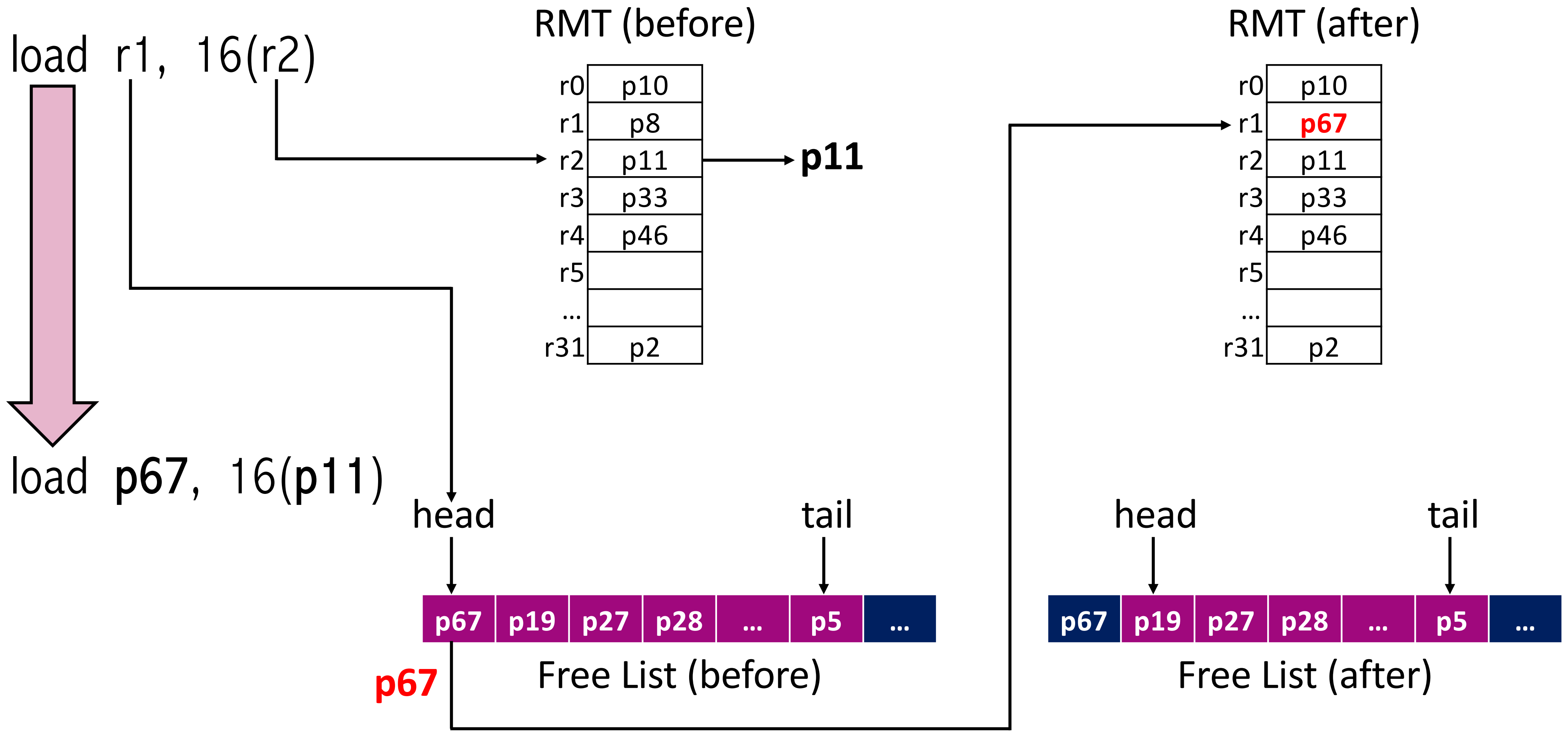
Logical source registers

- Obtain mapping from RMT

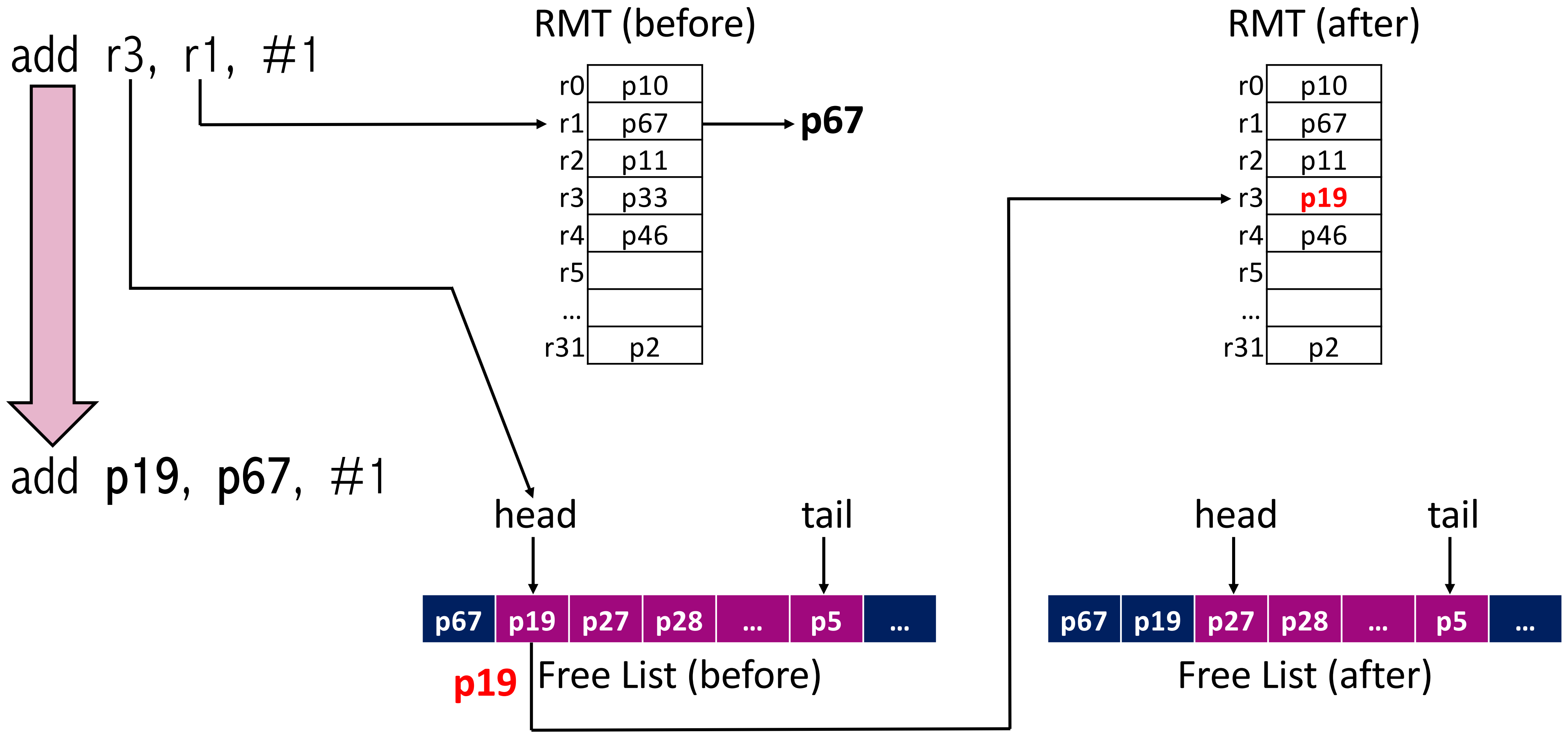
Logical destination registers

- Pop a free physical register from the free list
- Assign the physical register to the logical destination register
- Update RMT to reflect the new mapping

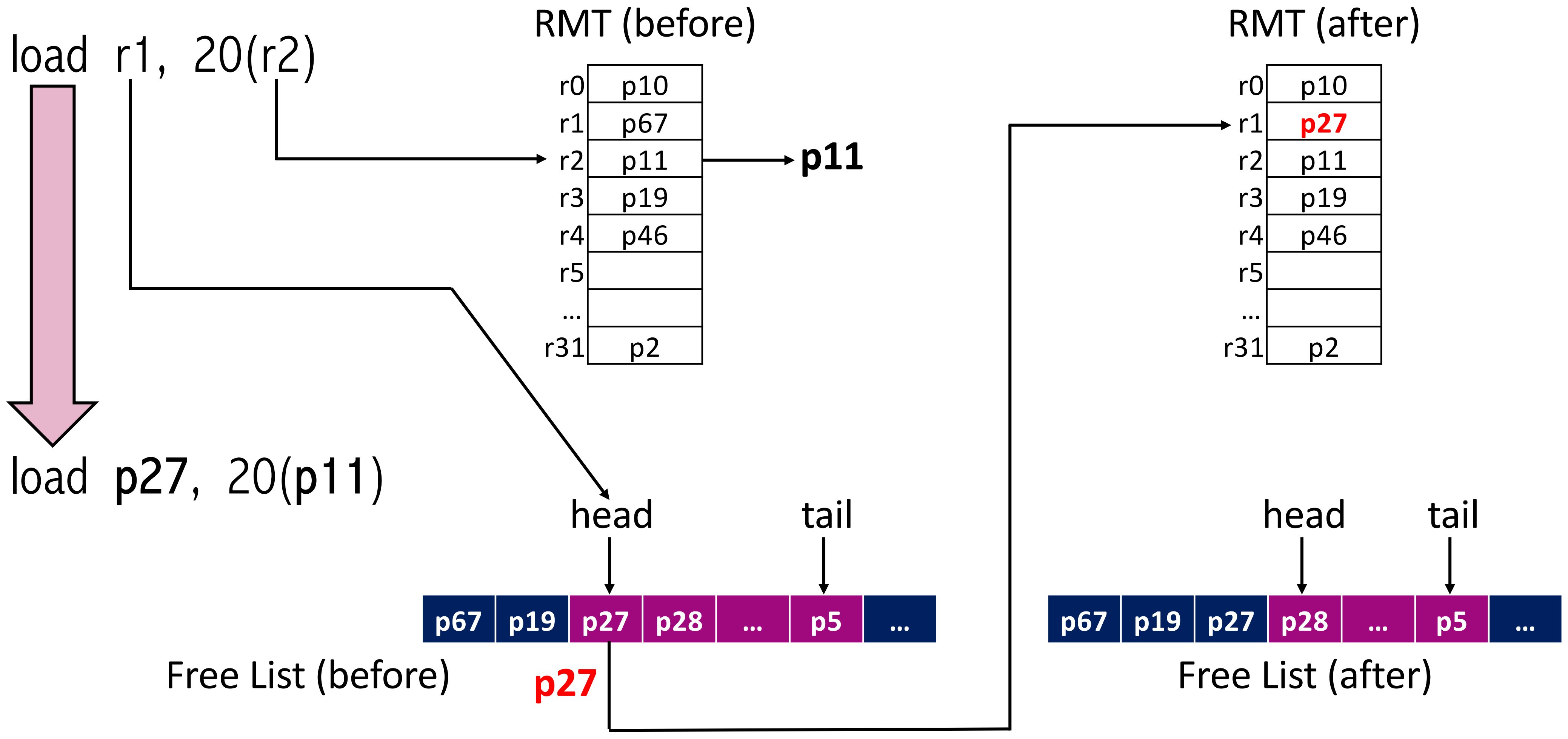
Renaming Example



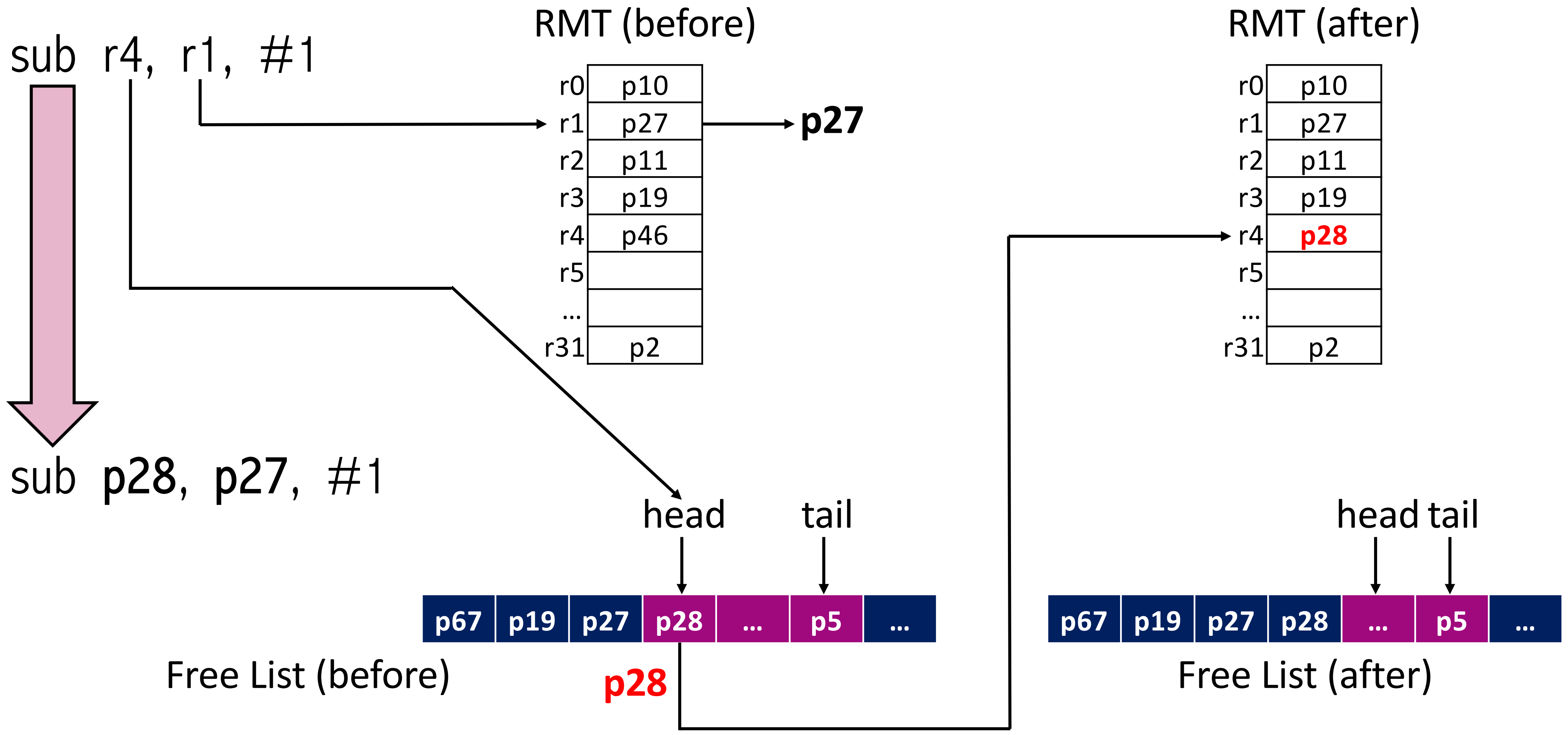
Renaming Example



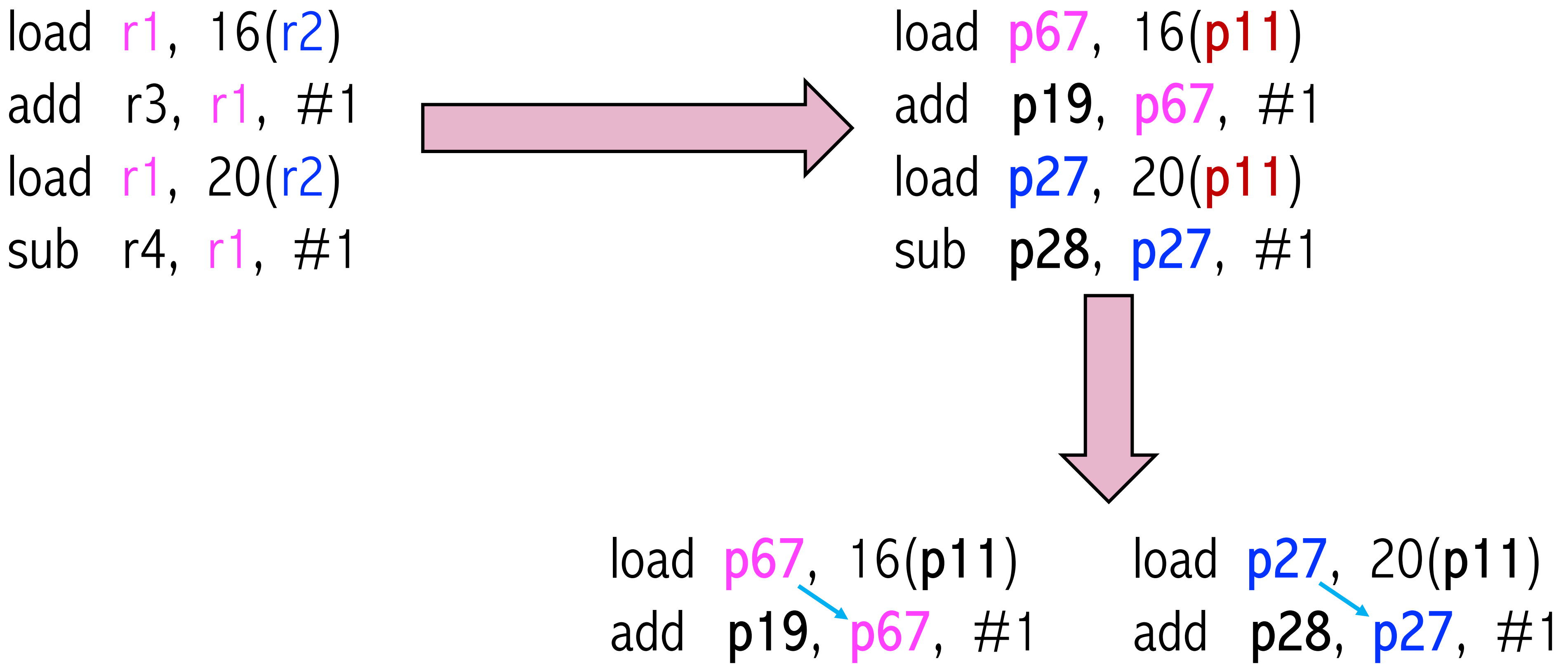
Renaming Example



Renaming Example



Renaming Example



```
load r1, 16(r2)
add  r3, r1, #1
load r1, 20(r2)
sub  r4, r1, #1
```

```
load p67, 16(p11)
add  p19, p67, #1
load p27, 20(p11)
sub  p28, p27, #1
```

```
load p67, 16(p11)        load p27, 20(p11)
add  p19, p67, #1        add  p28, p27, #1
```

Questions

- How/when to free registers?
- How to recover from exceptions and mispredictions?
- How to maintain the architectural state?
  - Subset of registers in PRF corresponding to the architectural state

Active List

- Active list
  - Contains the active instructions in program order
  - Active Instruction: Instruction that has been dispatched but not yet retired
  - Equivalent to the reorder buffer (ROB)
  - Other names: active window or instruction window

Active List Contents

Active List

		Current Mapping					
	Entry	Logical Dest.	Physical Dest.	Completed	Mispre-diction	Exception	PC
	1						
	2						
Head →	3	r1	p67	0	0	0	A
	4	r3	p19	0	0	0	B
	5	r1	p27	0	0	0	C
	6	r4	p28	0	0	0	D
Tail →	7						
	…						

Note: Do not confuse active list with the issue queue

Active List Operation

- Instruction dispatch
  - Reserve entry at tail
  - Initialize the entry
    - Write the instruction’s “current mapping” (logical and physical reg specifiers) and PC
    - Clear the completed, misprediction, and exception flags
  - Increment tail pointer

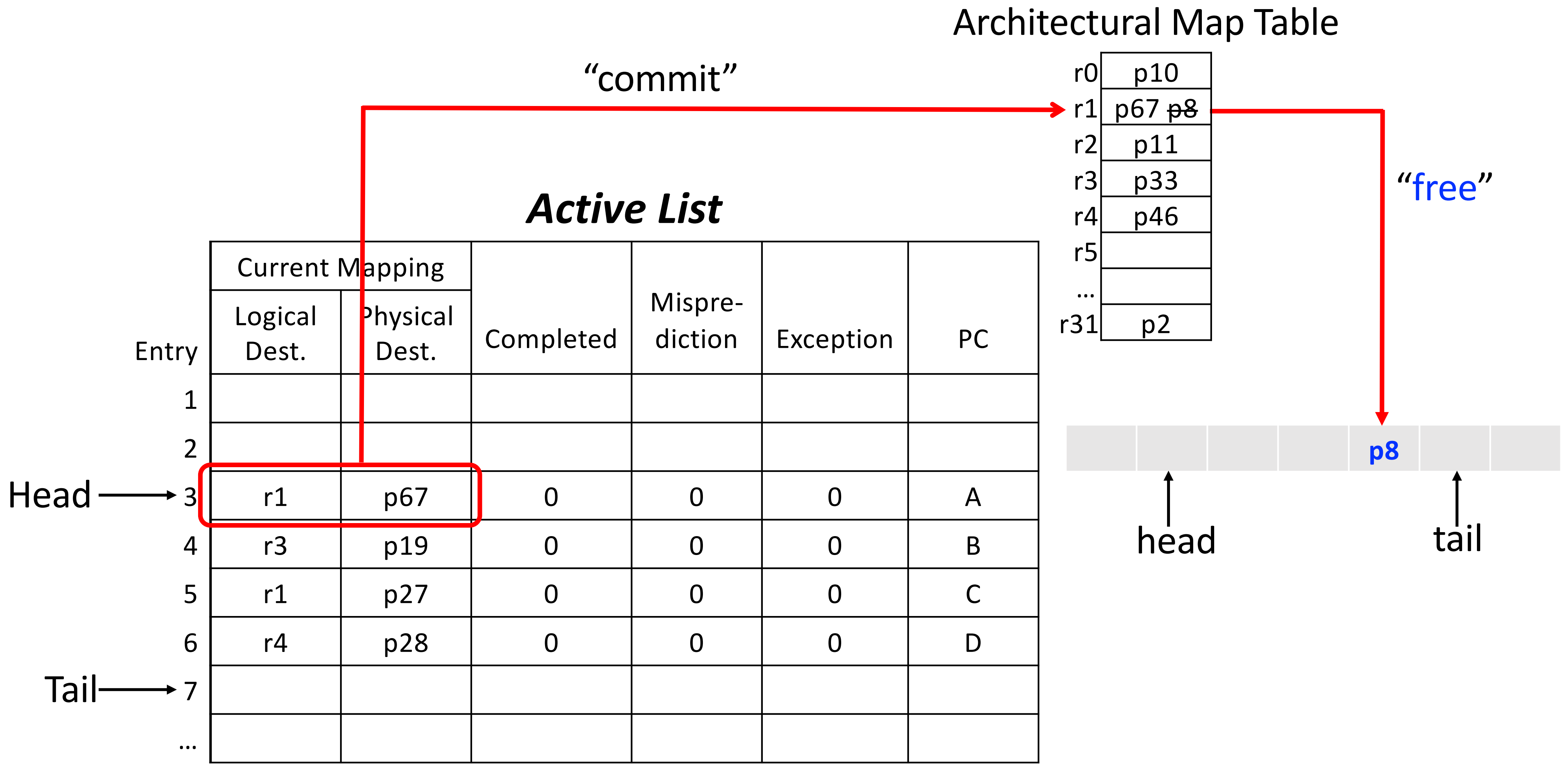
Active List Operation

- Instruction retirement
  - Instruction retirement frees the previously held “physical register” by the destination logical register
  - Retirement also commits the new value (the physical register specifier) of the instruction’s logical (destination) register
    - Need a new structure to store the committed mappings
    - Architectural Map Table (AMT)
    - Intel calls it Retirement Register Alias Table (RRAT)

Architectural Map Table (AMT)

- Architectural Map Table (AMT)
  - Contains the “committed” mapping of logical registers to physical registers
  - Note: In contrast with AMT, RMT contains the “speculative” mappings of logical to physical register identifiers
  - RMT is messy and dirty and should be discarded on mis-speculation
  - AMT is clean and must be preserved on an “exceptional” outcome
  - *AMT or RRAT enables possible recovery from mis-speculation and exceptions*

Committing & Freeing Registers



Architectural Map Table

r0	p10
r1	p67 ~~p8~~
r2	p11
r3	p33
r4	p46
r5	
…	
r31	p2

Active List

Entry	Current Mapping		Completed	Mispre-diction	Exception	PC
	Logical Dest.	Physical Dest.				
1						
2						
3	r1	p67	0	0	0	A
4	r3	p19	0	0	0	B
5	r1	p27	0	0	0	C
6	r4	p28	0	0	0	D
7						
…						

				p8		

Is AMT really needed?

- Having an RMT and AMT is costly
- We will see a redesign of PRF-style pipeline without AMT

Active List Operation (with AMT)

- Retire
  - Wait for instruction to reach the head (and complete)
  - Index the AMT using the head instruction’s logical destination register specifier
    - Free the “previous mapping” that is contained in the AMT
    - Push the previous mapping onto the free list
    - *Committing a new version implicitly means the old committed version is no longer needed for (potential) recovery*
    - Physical register indicated by “current mapping” is committed by rewriting the AMT with the current mapping
  - Increment head pointer to commit younger instructions in the active list (deplete the instruction window)

Handling Exception and Mis-speculation

- Offending instruction sets its exception/misprediction bit in the active list

- When offending instruction reaches the head of the active list
  - Squash active list: Set Head = Tail
  - Squash/flush pipeline: Squash all instructions in front-end stages, IQ, functional units, etc.
  - Restore RMT to committed state: **Copy AMT to RMT**
  - Restore free list: Push mappings of all instructions after the offending one back onto the free list
  - Save PC of offending instruction (get it from the head of active list)
    - EPC in MIPS ISA where handler will return to
    - EPC = AL[al_head].PC
  - Trap to exception handler
    - PC = address of exception handler

Implementation Challenge 1: Pipeline Squash

- Squashing instructions in the pipleine
  - Global squash signal a cycle time bottleneck
    - Physical VLSI design-level details (interconnects)
    - Extending to multiple cycles provides some relief
  - Alternative (no global squash signal)
    - What if we allow all in-flight instructions to complete?
    - A.k.a. draining the pipleine
    - Do not accept (fetch) new instructions
    - Guard the AMT (i.e., set write enable to zero)
    - Can overlap some cycles with restoring RMT from AMT if it takes multiple cycles to copy

Challenge 2: Restoring RMT from AMT

- Solution # 1
  - Flash copy: Customized circuit that enables single-cycle copying
  - Requires sophisticated SRAM design
- Solution # 2
  - Serial copy: Conventional SRAM design
  - Speed limited by # read and # write ports
  - # cycles to restore RMT =

$$\#\,cycles\ to\ restore\ RMT = \#\,logical\ registers \big/ \min(\#\,AMT\ read\ ports, \#\,RMT\ write\ ports)$$

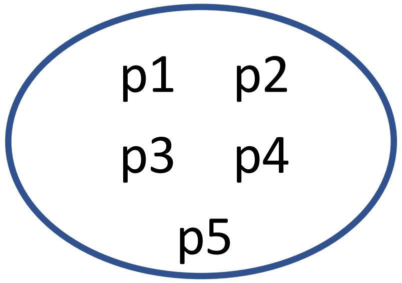
Free List Example

Logical registers: r1

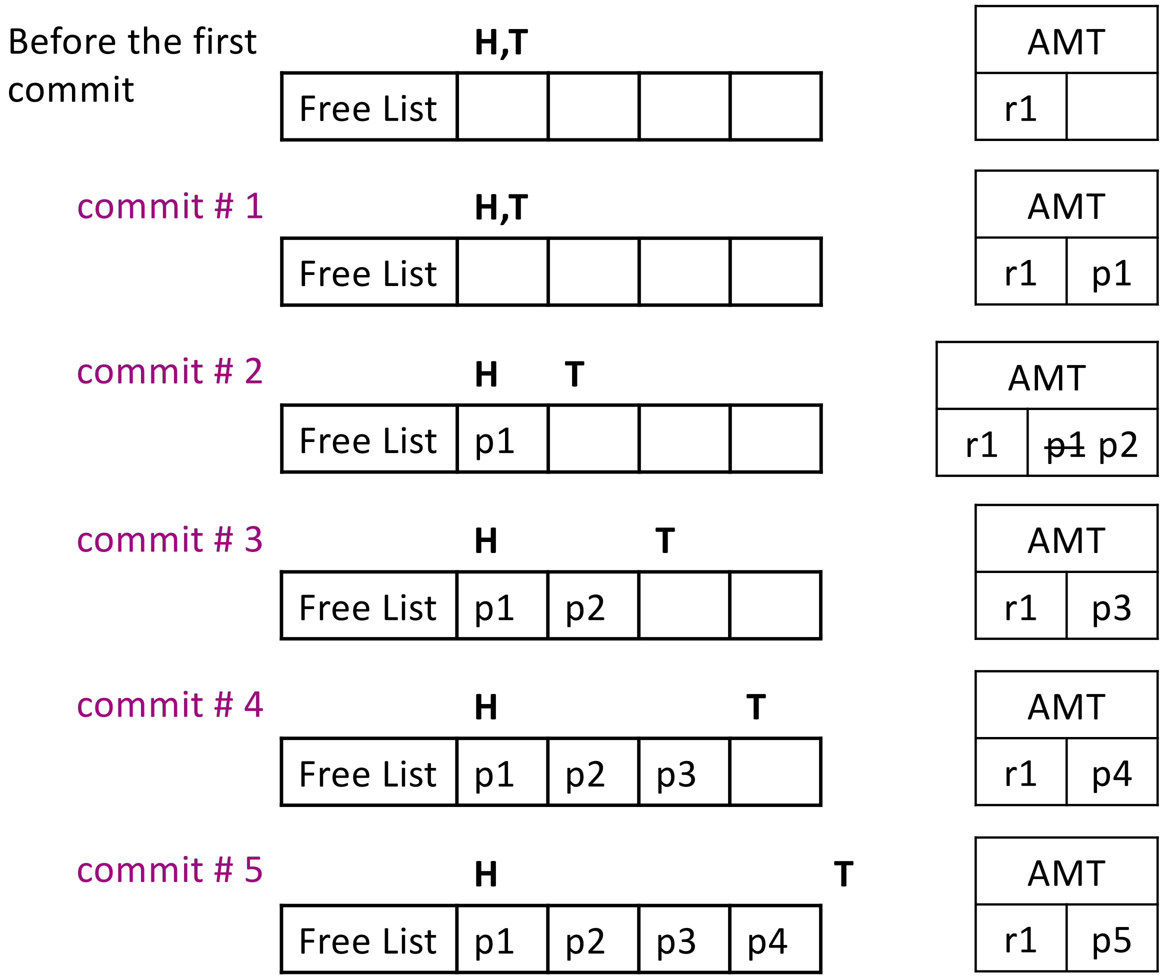
Physical registers: p1-p5

# AMT entries: 1

initial pool of free registers



Scenario: *Dispatch five instructions each redefining r1 by grabbing a free register from the initial pool. Then committ instructions from the head of the active list one by one*



Active list (not shown) is now empty

Backup: AL snapshot before first commit

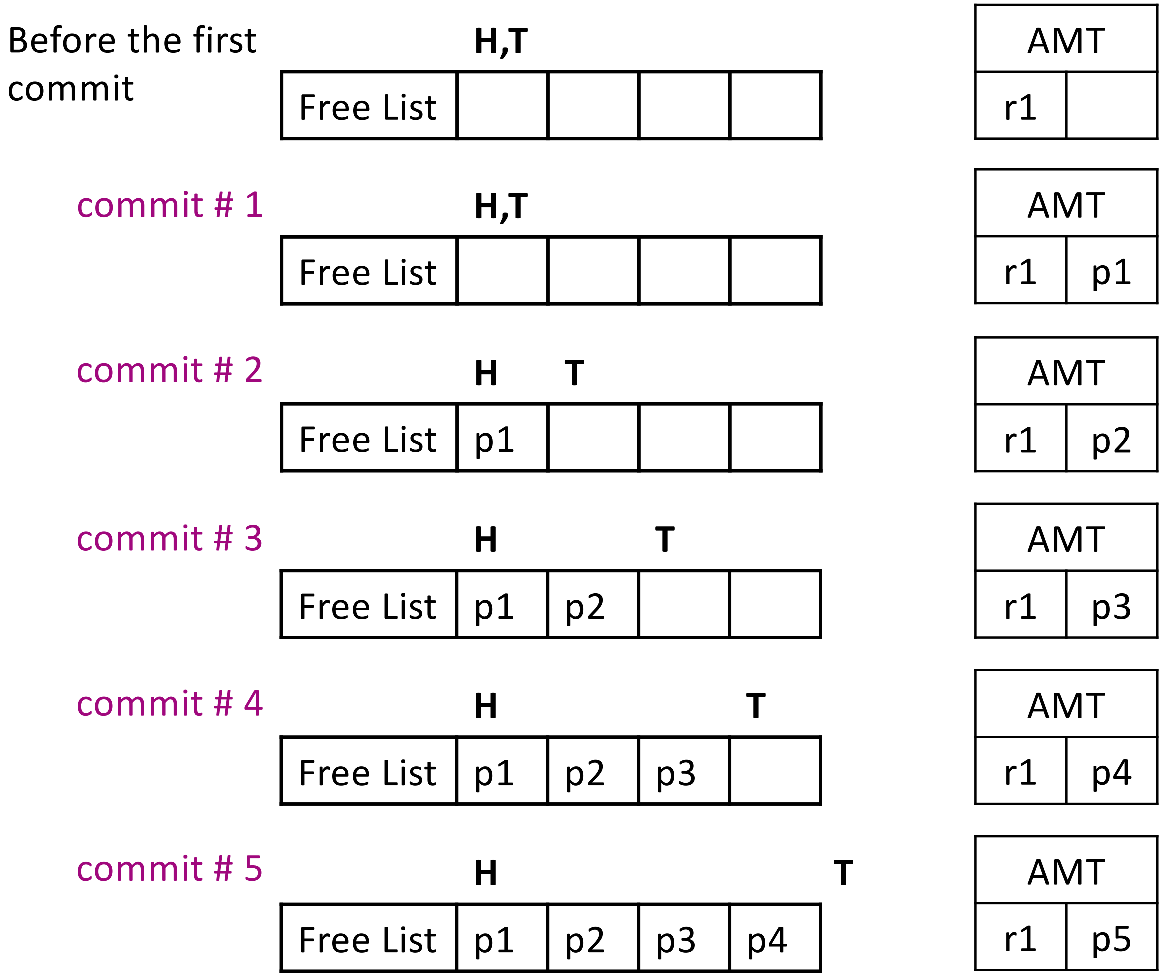
Active List

	Entry	**Current Mapping**		Completed	Mispre-diction	Exception	PC
		Logical Dest.	Physical Dest.				
	1						
head →	2	r1	p1	0	0	0	A
	3	r1	p2	0	0	0	B
	4	r1	p3	0	0	0	C
	5	r1	p4	0	0	0	D
	6	r1	p5	0	0	0	E
Tail →	7						
	…						

Free List Example

Note: *The implementation must distinguish b/w empty and full free list (circular FIFO)*

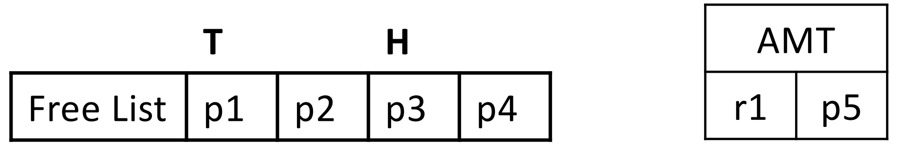
Question: *What is between head and tail?*
Answer: Speculative, free registers



Active list (not shown) is now empty

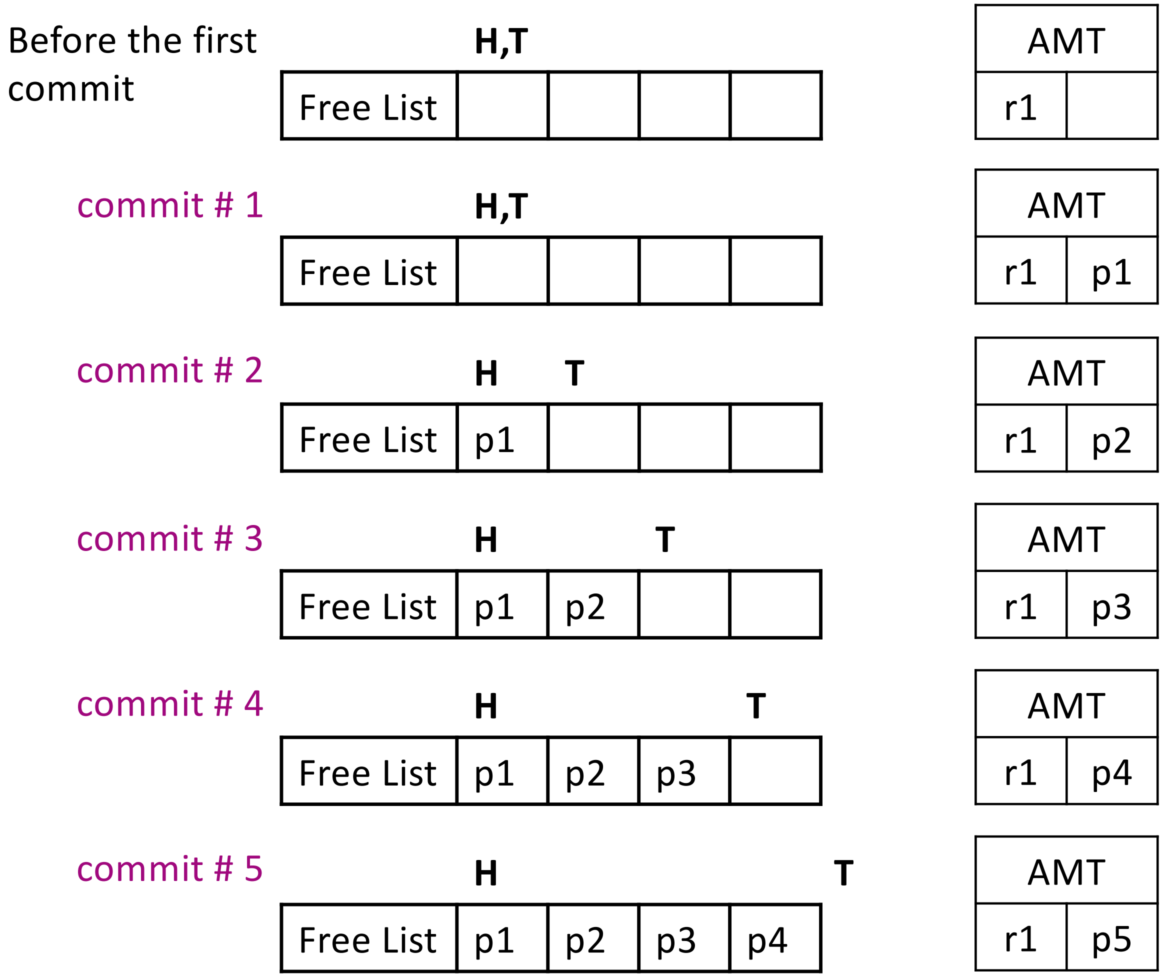
Free List Example

New Scenario: *After commit # 5, dispatch two instructions each renaming r1 by grabbing a free register from the list.*



Question: *What is between tail and head?*

Answer: speculative registers, allocated (dispatched) but not committed

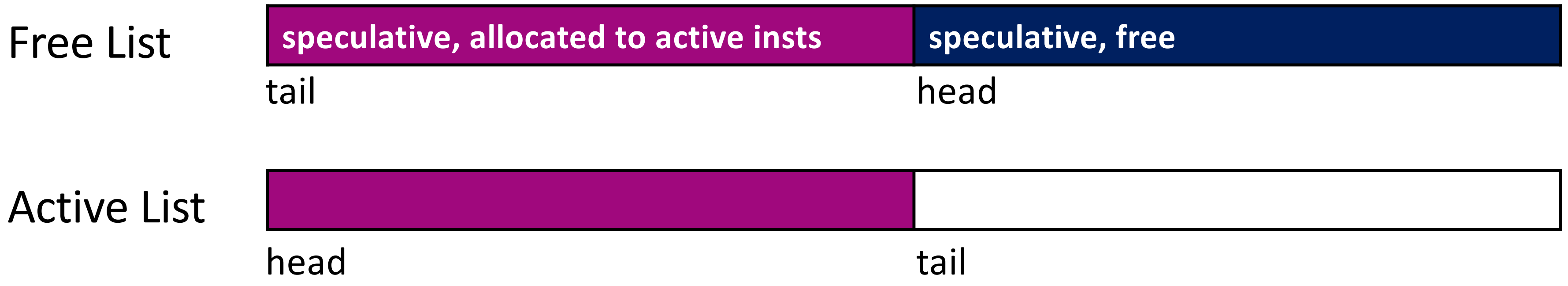


Active list (not shown) is now empty

Restoring Free List

Question: *What is between head and tail?*
Answer: Speculative, free registers

Question: *What is between tail and head?*
Answer: speculative registers, allocated (dispatched) but not committed



Question: *Is there a simple way to restore/repair the free list?*

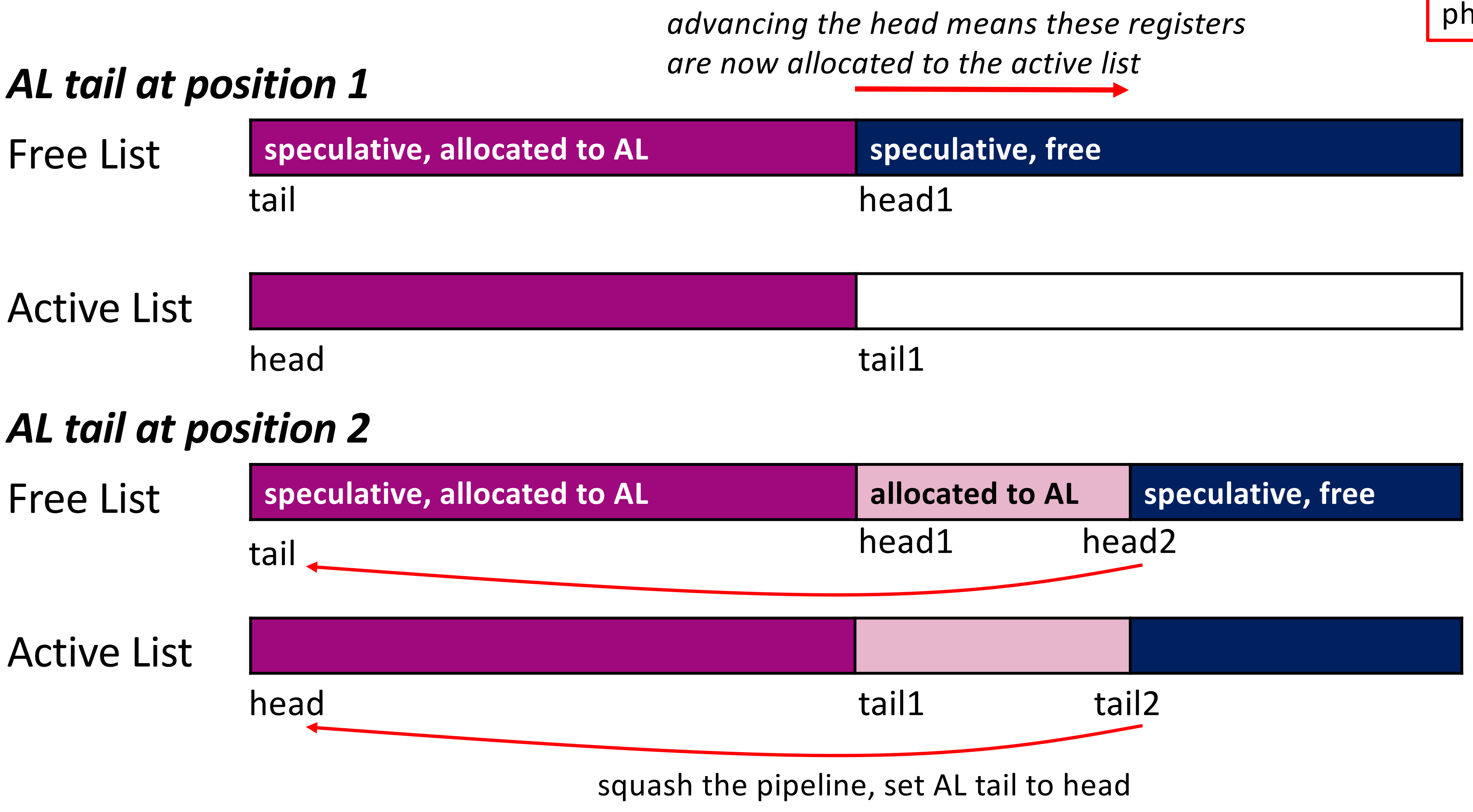
Challenge 3: Restoring Free List

- Solution # 1
  - Slow and complex: Scan active list between head and tail, pusing each instruction’s current mapping onto the free list

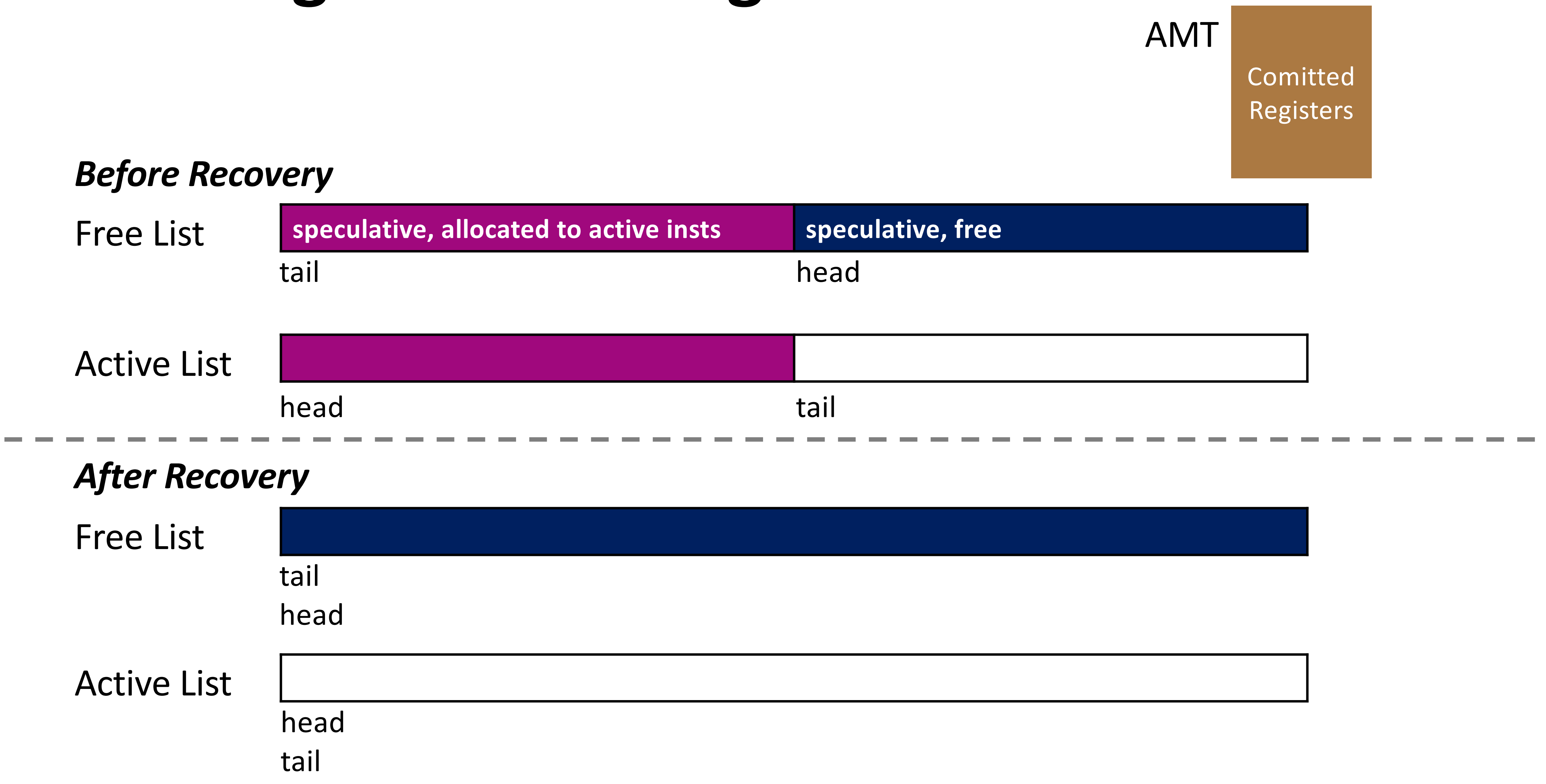
- Solution # 2
  - Fast and simple
  - Observations about free list contents
    - Between H and T: List of free physical registers
    - Between T and H: List of physical registers allocated to instructions between active list H and T
  - Restoring free list simply by rolling back head pointer to tail pointer and noting that free list is full
    - Head = Tail and freelist_full = true
    - Circular FIFO must distinguish b/w empty and full cases

Challenge 3: Restoring Free List

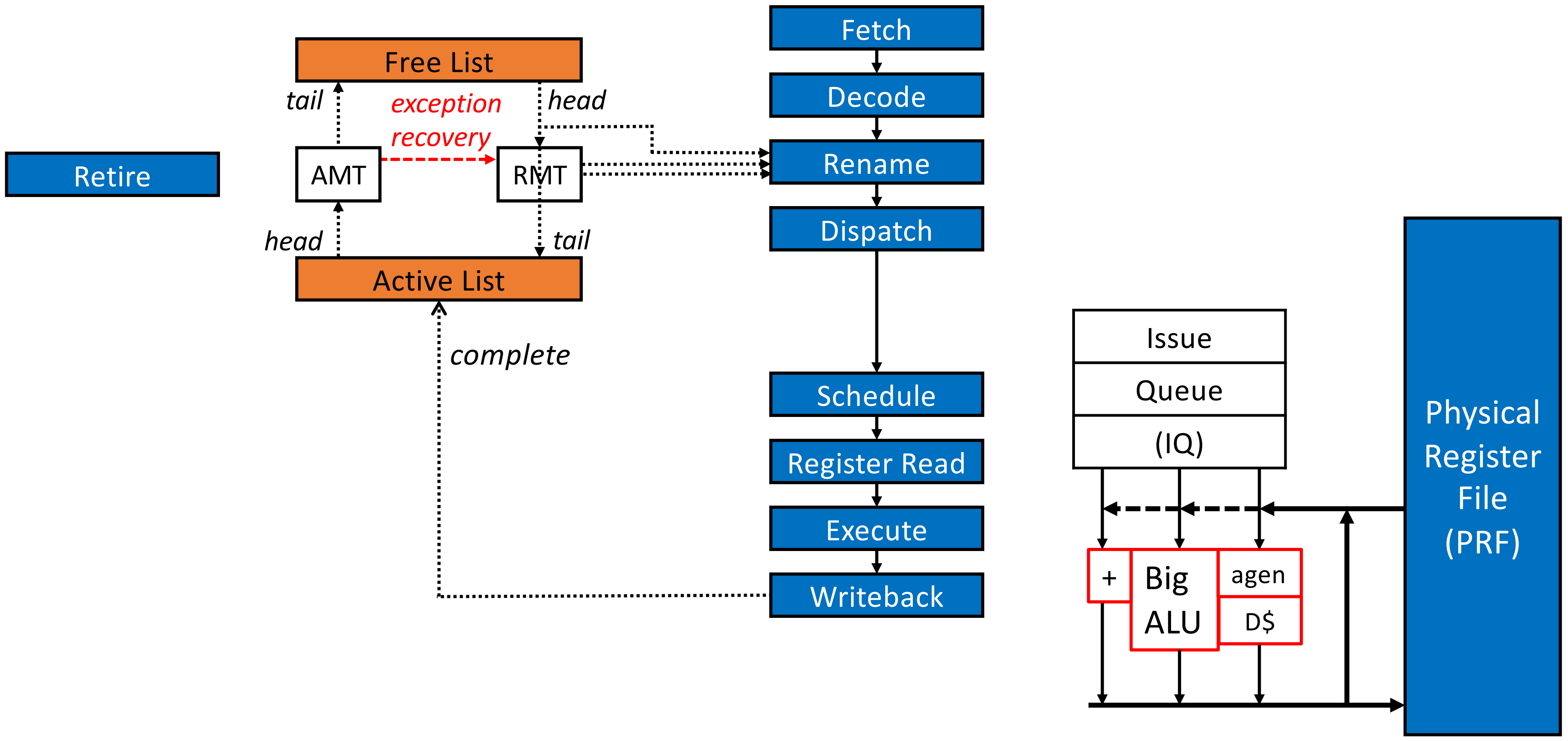
Sidenote: AL has current logic to physical mappings



Challenge 3: Restoring Free List



Contemporary Superscalar Microarchitecture



Remember the different names

- Architectural Register File (ARF)
    - Retirement Register File (RRF)
- Rename Map Table (RMT)
    - Register Alias Table (RAT)
    - Front-End Map Table (when used in PRF)
- Architectural Map Table (AMT)
    - Reteirement Register Alias Table (RRAT)
    - Backend Map Table
    - Used only in PRF for fast recovery

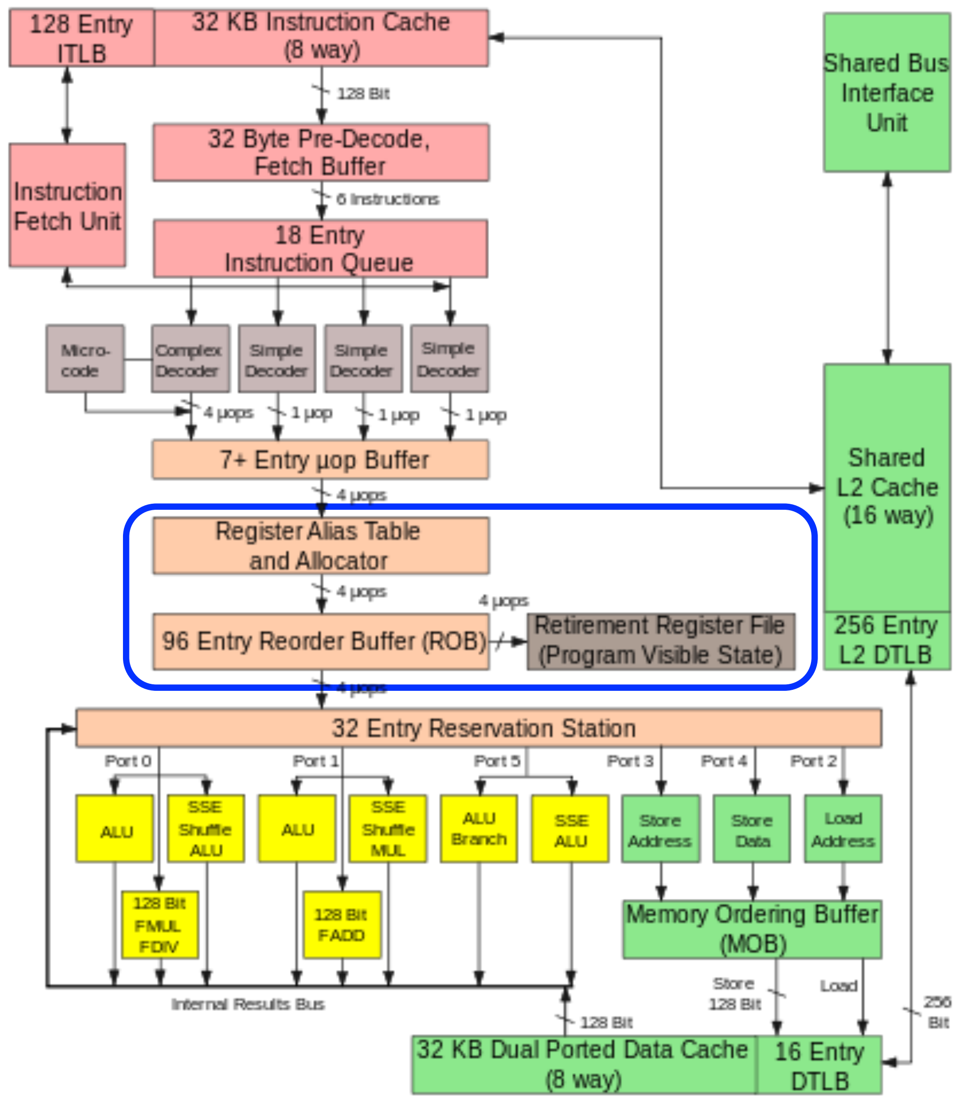
Remember the different names

- Reservation Stations
  - Instruction Queue
  - Scheduling Window
- Active List (AL) or Reorder Buffer (ROB)
  - Instruction window
  - Active window

State-of-affairs: PRF

- ARF+ROB
  - P6 through Core 2 Duo (Merom), Nehalem
- PRF (nearly every high-performance processor today)
  - Pentium 4 (P68), Sandybridge, and later
  - AMD Bulldozer, Bobcat
  - MIPS R10000 (**basis of today's lecture slides**)
  - IBM Power4, Power5, Power_6,7,8,9_?
  - Alpha 21264
- Recent regression
  - Intel Silvermont, ARM Cortex A15

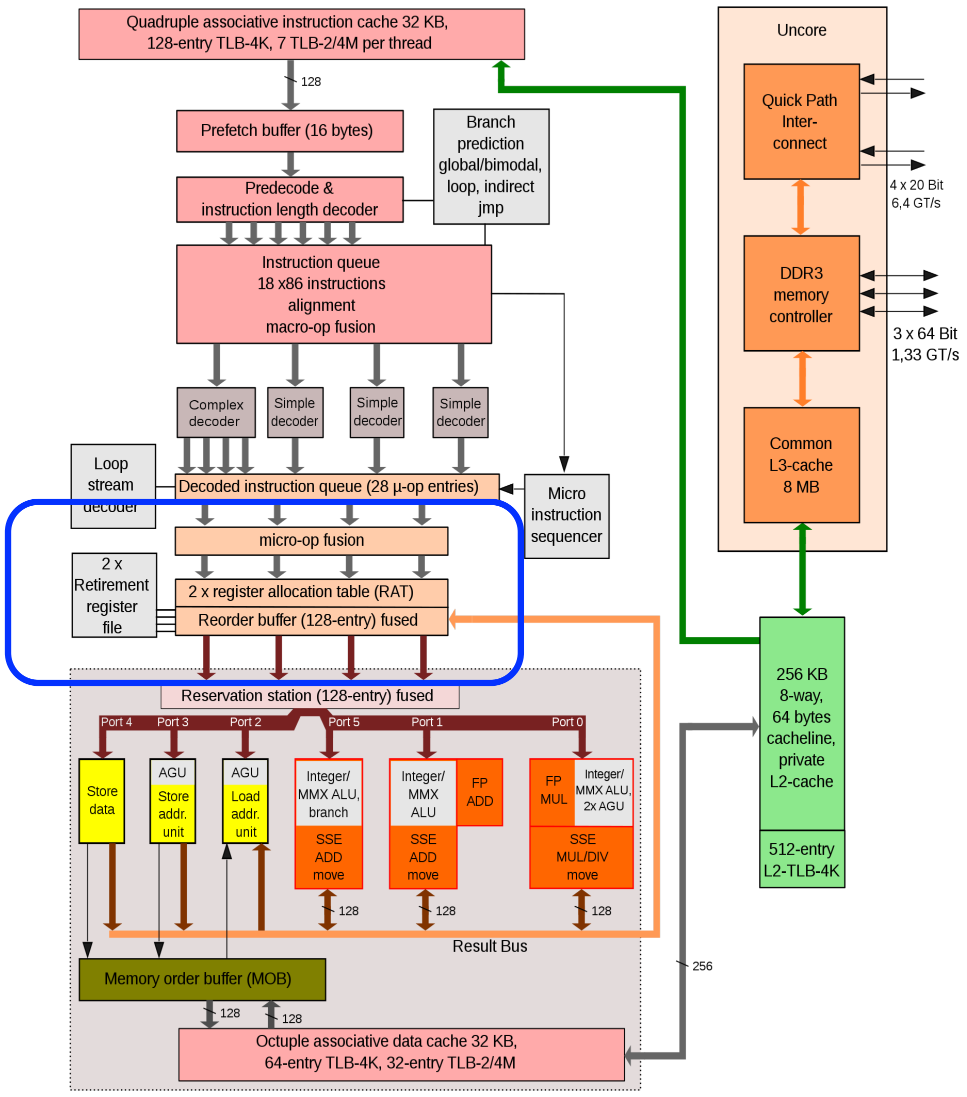
Intel Yonah
2006
Core 2



Intel Core 2 Architecture

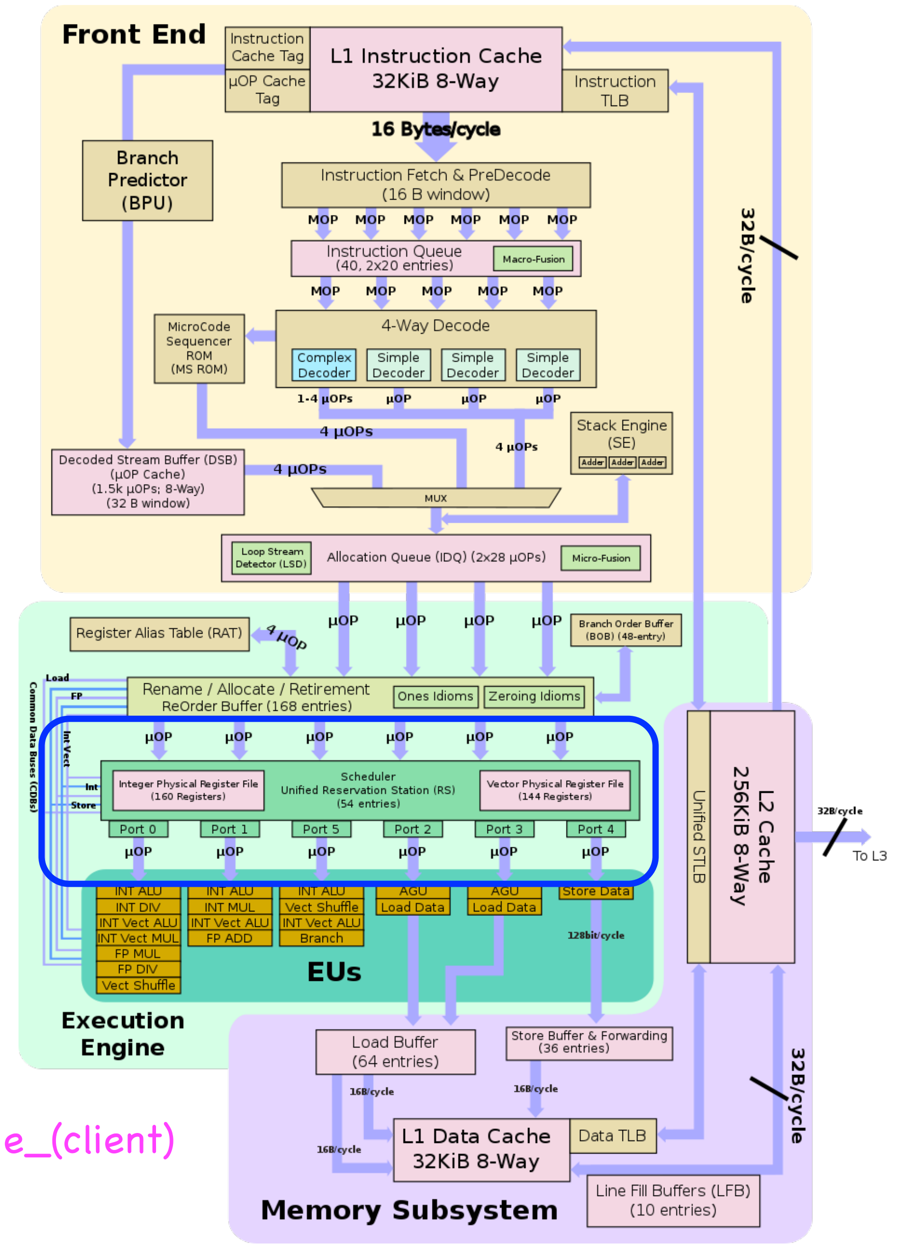
Intel Nehalem 2008 Core i5, i7

Intel Nehalem microarchitecture


Quadruple associative instruction cache 32 KB, 128-entry TLB-4K, 7 TLB-2/4M per thread
128
Prefetch buffer (16 bytes)
Branch prediction global/bimodal, loop, indirect jmp
Predecode & instruction length decoder
Instruction queue 18 x86 instructions alignment macro-op fusion
Complex decoder
Simple decoder
Simple decoder
Simple decoder
Loop stream decoder
Decoded instruction queue (28 µ-op entries)
Micro instruction sequencer
micro-op fusion
2 x Retirement register file
2 x register allocation table (RAT)
Reorder buffer (128-entry) fused
Reservation station (128-entry) fused
Port 4
Port 3
Port 2
Port 5
Port 1
Port 0
Store data
AGU
Store addr. unit
AGU
Load addr. unit
Integer/ MMX ALU, branch
SSE ADD move
Integer/ MMX ALU
SSE ADD move
FP ADD
FP MUL
Integer/ MMX ALU, 2x AGU
SSE MUL/DIV move
128
128
128
Result Bus
Memory order buffer (MOB)
128
128
Octuple associative data cache 32 KB, 64-entry TLB-4K, 32-entry TLB-2/4M
256
Uncore
Quick Path Inter-connect
4 x 20 Bit 6,4 GT/s
DDR3 memory controller
3 x 64 Bit 1,33 GT/s
Common L3-cache 8 MB
256 KB 8-way, 64 bytes cacheline, private L2-cache
512-entry L2-TLB-4K

GT/s: gigatransfers per second

Intel Sandy Bridge
2010
Core i5, i7



Front End
Instruction Cache Tag
µOP Cache Tag
L1 Instruction Cache 32KiB 8-Way
Instruction TLB
16 Bytes/cycle
Branch Predictor (BPU)
Instruction Fetch & PreDecode (16 B window)
MOP
Instruction Queue (40, 2x20 entries)
Macro-Fusion
MicroCode Sequencer ROM (MS ROM)
4-Way Decode
Complex Decoder
Simple Decoder
1-4 µOPs
µOP
4 µOPs
Stack Engine (SE)
Adder
Decoded Stream Buffer (DSB) (µOP Cache) (1.5k µOPs; 8-Way) (32 B window)
MUX
Loop Stream Detector (LSD)
Allocation Queue (IDQ) (2x28 µOPs)
Micro-Fusion
32B/cycle
Register Alias Table (RAT)
4 µOP
Branch Order Buffer (BOB) (48-entry)
Common Data Buses (CDBs)
Load
FP
Int Vect
Int
Store
Rename / Allocate / Retirement ReOrder Buffer (168 entries)
Ones Idioms
Zeroing Idioms
Integer Physical Register File (160 Registers)
Scheduler Unified Reservation Station (RS) (54 entries)
Vector Physical Register File (144 Registers)
Port 0
Port 1
Port 5
Port 2
Port 3
Port 4
INT ALU
INT DIV
INT Vect ALU
INT Vect MUL
FP MUL
FP DIV
Vect Shuffle
INT MUL
FP ADD
Branch
AGU
Load Data
Store Data
128bit/cycle
EUs
Execution Engine
Unified STLB
L2 Cache 256KiB 8-Way
32B/cycle
To L3
Load Buffer (64 entries)
Store Buffer & Forwarding (36 entries)
16B/cycle
L1 Data Cache 32KiB 8-Way
Data TLB
Line Fill Buffers (LFB) (10 entries)
Memory Subsystem

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

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.

Alternative Design: No AMT

Instead of the current logical-physical mapping, remember the previous mapping in the active list

Active List

		★ Previous Mapping					
	Entry	Logical Dest.	Physical Dest.	Completed	Mispre-diction	Exception	PC
	1						
	2						
Head →	3	r1	p67	0	0	0	A
	4	r3	p33	0	0	0	B
	5	r1	p8	0	0	0	C
	6	r4	p46	0	0	0	D
Tail →	7						
	…						

Key Idea: On a misspeculation we can rollback the state of the RMT by walking the AL from tail to head

Active List Operation (No AMT)

Rename

- Read out the previous mapping (from the RMT) for the logical destination register (extra read port)
- Update the RMT with the new mapping

Approach # 1 (with AMT)

```
phys. src. reg. 1 = RMT[logical src. reg. 1] //RMT read port
phys. src. reg. 2 = RMT[logical src. reg. 2] //RMT read port

phys. dest. reg. = pop new mapping from free list
RMT[logical dest. reg.] = phys. dest. reg. //RMT write port
```

Approach # 2 (without AMT)

```
phys. src. reg. 1 = RMT[logical src. reg. 1] //RMT read port
phys. src. reg. 2 = RMT[logical src. reg. 2] //RMT read port
previous mapping = RMT[logical dest. reg.] //extra RMT read port

phys. dest. reg. = pop new mapping from free list
RMT[logical dest. reg.] = phys. dest. reg. //RMT write port
```

Active List Operation (No AMT)

Instruction dispatch

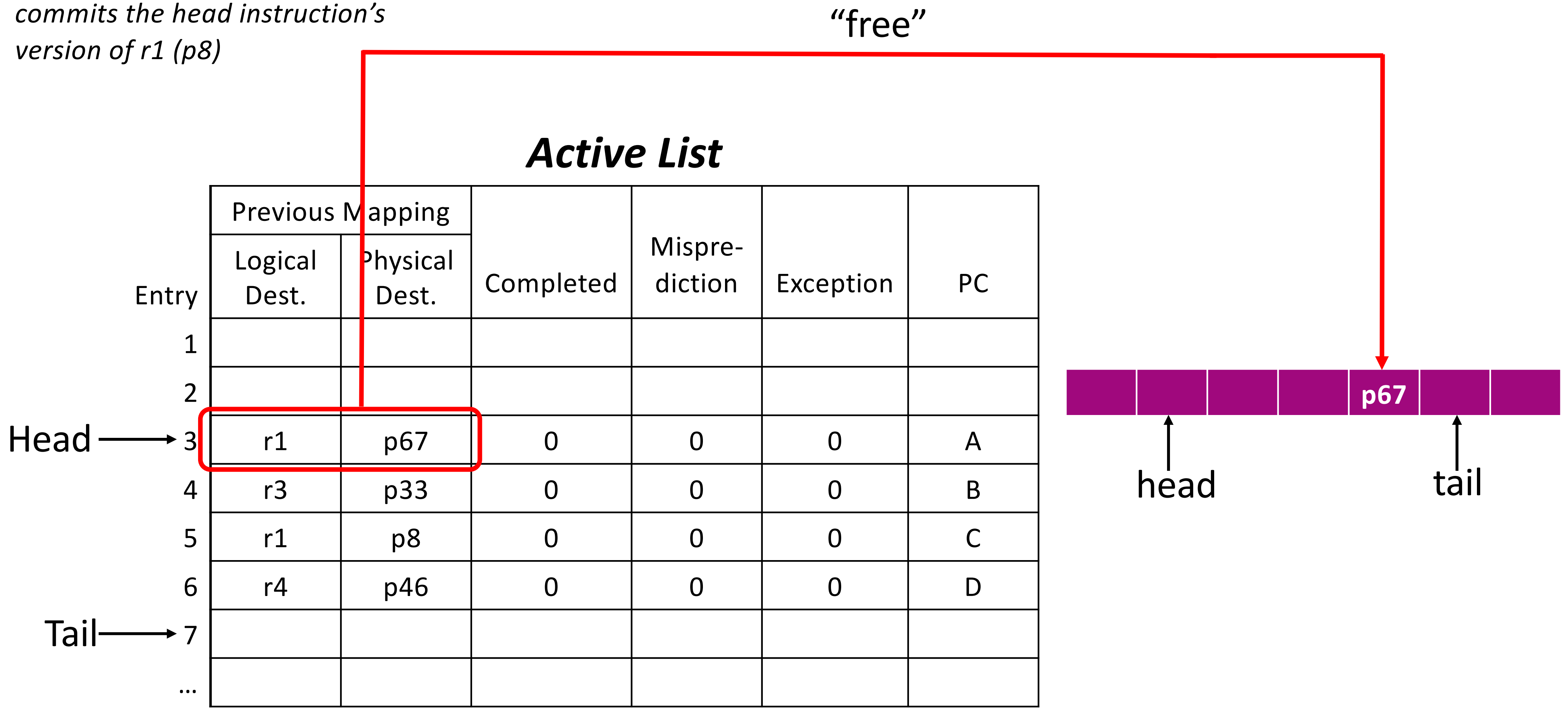
- Reserve entry at tail
- Initialize the entry
  - Write the instruction’s “previous mapping” (logical destination register specifier and the previous mapping on that logical register) and PC
  - Reset the completed, misprediction, and exception flags
- Increment tail pointer

Active List Operation (No AMT)

- Instruction retirement
  - Wait for the head instruction to complete
  - Push its previous mapping onto the free list. This has two implicit effects:
    - The prior committed version of the instruction’s logical destination register is freed
    - The instruction’s physical destination register is committed (implicitly)
  - Increment head pointer

Committing & Freeing Registers

“commit”: *This action implicitly commits the head instruction’s version of r1 (p8)*



Active List

		Previous Mapping					
	Entry	Logical Dest.	Physical Dest.	Completed	Mispre-diction	Exception	PC
	1						
	2						
Head	3	r1	p67	0	0	0	A
	4	r3	p33	0	0	0	B
	5	r1	p8	0	0	0	C
	6	r4	p46	0	0	0	D
Tail	7						
	…						

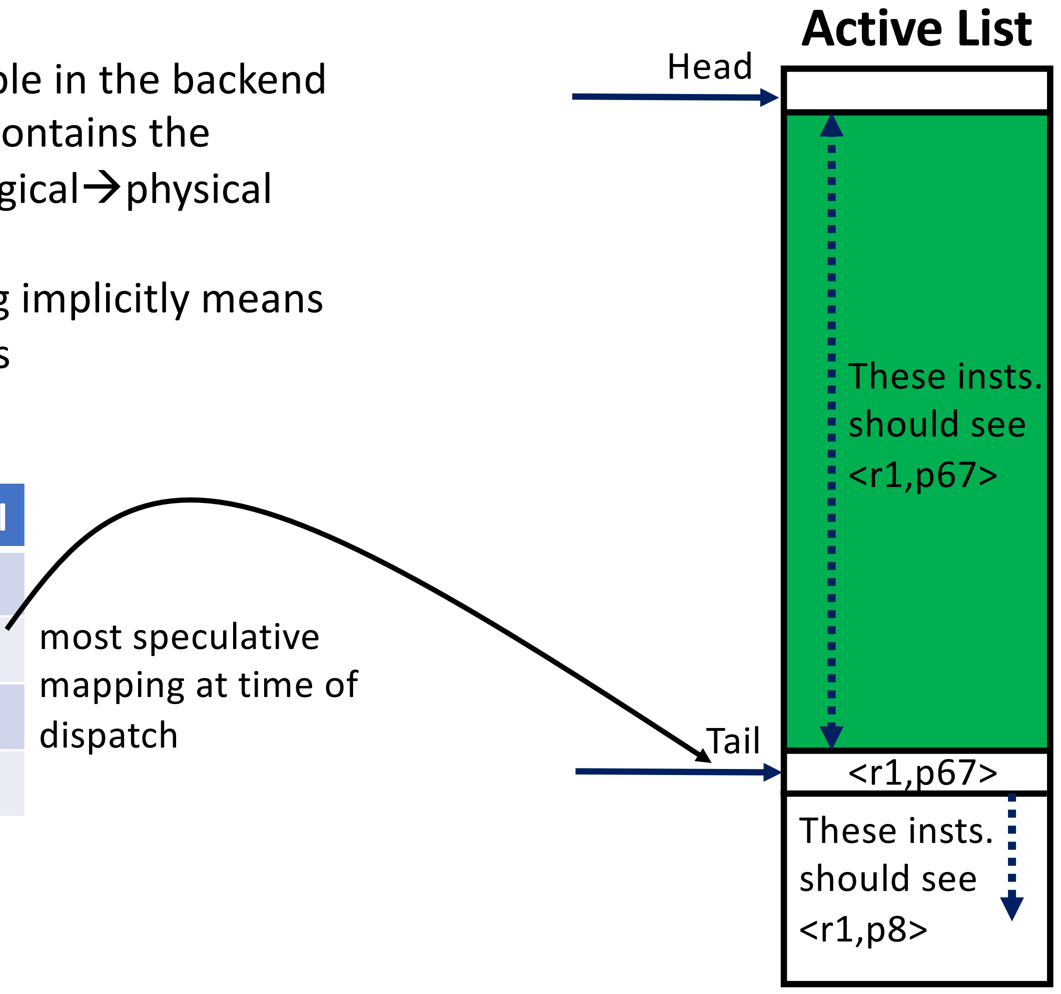
Freeing Registers in No-AMT Design

Background

- No map table in the backend
- Active list contains the previous logical→physical mappings
- Committing implicitly means freeing regs

RMT

Logical	Physical
r1	~~p67~~ p8


most speculative mapping at time of dispatch
Active List
Head
Tail
These insts. should see <r1,p67>
<r1,p67>
These insts. should see <r1,p8>

Freeing Registers in No-AMT Design

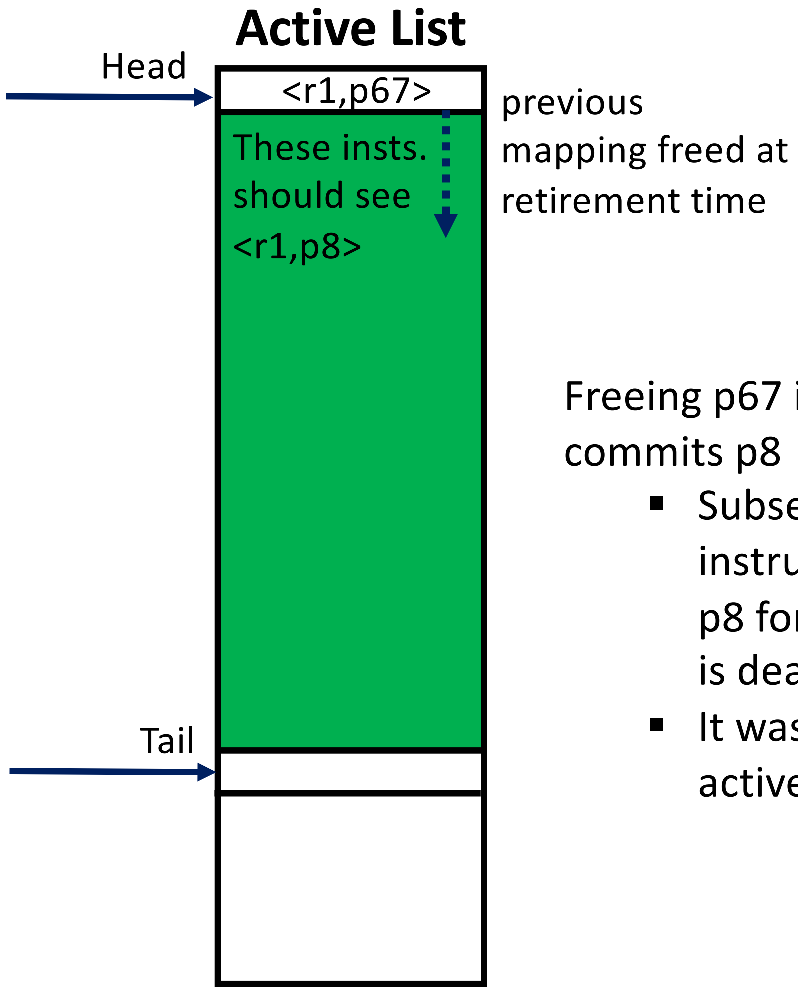
Retirement of <r1,p67> from head

- p8 busy bit in the busy table is 0 (if the instruction is complete)
- All subsequent insts observe <r1,p8> (unless another inst. renames r1)

RMT

Logical	Physical
r1	~~p67~~ p8

most speculative mapping at time of dispatch


Active List
Head
<r1,p67>
previous mapping freed at retirement time
These insts. should see <r1,p8>
Tail

Freeing p67 implicitly commits p8

- Subsequent instructions should see p8 for r1 because p67 is dead
- It was kept in the active list for recovery

Committing & Freeing Registers

“commit”: *This action implicitly commits the head instruction’s version of r1 (p67)*

- This is how you should look at it:
  - RMT had the non-sepculative mapping (r1, p67)
  - An instruction K redefines the mapping to (r1, p8), RMT now contains (r1,p8)
    - (r1, p8) is the speculative version of mapping
  - The instruction K stores the previous mapping (r1, p67) in AL
  - Now, think, what is the meaning of freeing (committing) p67 at retirement
    - That r1 is no longer mapped to p67 (r1→p67 is dead)
    - That subsequent instructions in program order should see r1→p8
    - And if there is an exception, we need to restore RMT to r1→p8
    - Instruction with (r1, p8) is committing so it is no longer speculative

Rollback-Based Recovery (Scenario # 1)

- Handling exceptions and misspeculation
  - To repair RMT without AMT, we need rollback-based recovery
  - No map table in the backend so can’t flash copy AMT into RMT
  - But, active list contains the previous logical→physical mappings
  - Remember: Committing implicitly updates the arch. state/mapping
  - Misspeculation means we are not going to commit instructions so RMT needs to be brought into a precise state

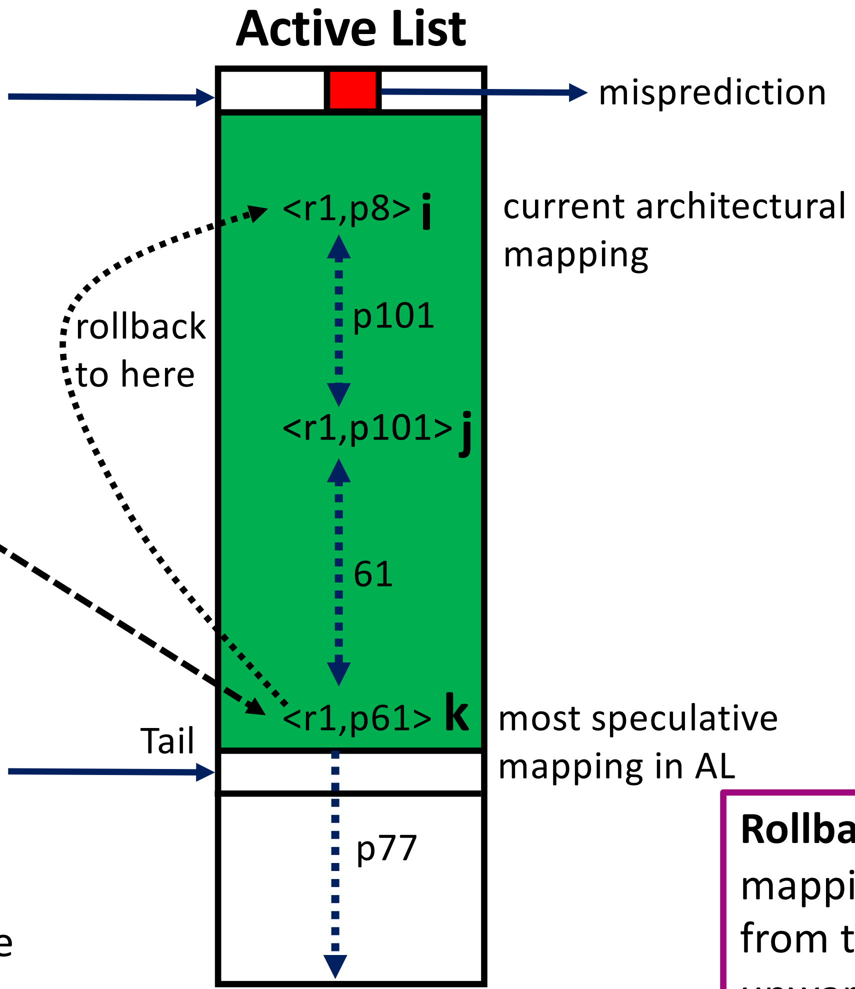
- We will need rollback-based recovery in other scenarios
  - Try to grasp the general concept
  - You will apply the concept to solve some interesting problems

Rollback-Based Recovery

RMT (before)

Logical	Physical
r1	~~p61~~ p77

most speculative mapping



RMT after Recovery

Logical	Physical
r1	p8

Committing <r1,p8> will make p101 as the architectural mapping

- But instruction holding <r1,p8> never commits
- So we want RMT to reflect the correct architectural state
- Rollback from Tail to Head

Rollback: Start putting previous mappings for r1 into RMT starting from tail and incrementally moving upwards

Handling Exception and Mis-speculation

- Offending instruction sets its exception/misprediction bit in the active list

- When offending instruction reaches the head of the active list
  - Squash active list: Set Head = Tail
  - Squash/flush pipeline: Squash all instructions in front-end stages, IQ, functional units, etc.
  - Restore RMT to committed state: Scan the active list *backward* from tail to head, restore previous mappings into RMT
  - Restore free list: (1) Head = Tail (easy) or (2) While restoring RMT, “undone” current mappings are pushed back onto free list
  - Save PC of offending instruction (get it from the head of active list)
  - Trap to exception handler

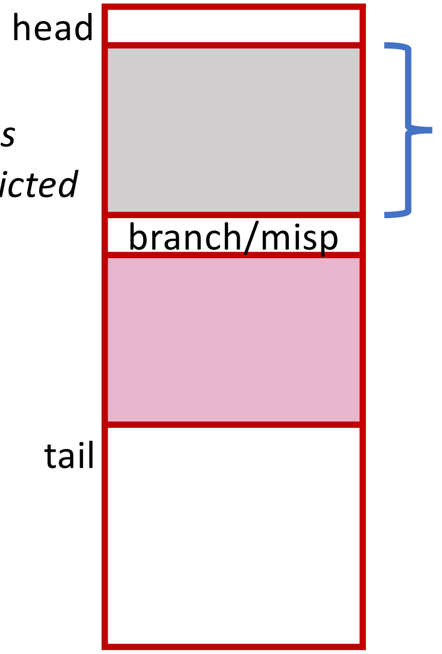
Branch Misprediction Recovery

- Drawback of approaches we have discussed so far
  - Wait for the mispredicted branch to reach the head of the active list before initiating recovery
  - Do not discriminate between exceptions and mispredictions
  - **Today:** Approaches that initiate recovery as soon as the misprediction is discovered (as soon as the branch instruction executes), from the middle of the active list

Recovery from the middle of AL

The next two approaches initiate recovery from branch mispredictions as soon as the misprediction is discovered (as soon as the branch instruction executes), from the middle of the active list

Assuming there are no unresolved branches b/w head and mispredicted branch



By waiting for the branch to reach the head, and then initiate recovery, we miss an opportunity to exploit parallelism: execute instructions b/w head and branch and recover from branch misprediction in parallel

Note: Instructions b/w branch and head are already dispatched, do not care about changes to RMT and free list

Recovery from the middle of AL

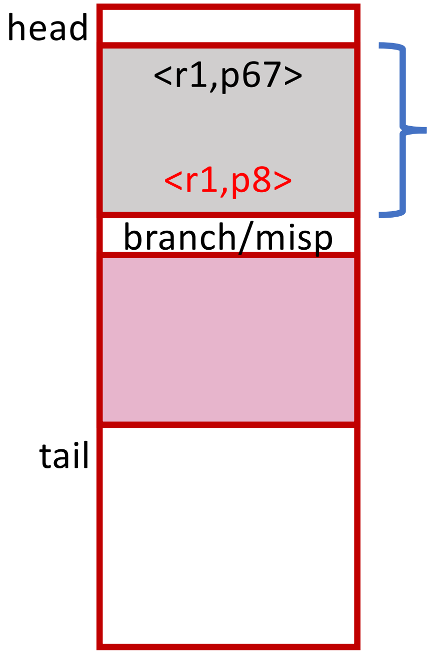
The next two approaches initiate recovery from branch mispredictions as soon as the misprediction is discovered (as soon as the branch instruction executes), from the middle of the active list

Free List: For both approaches, checkpoint the free list head pointer when the branch is renamed. Restoring the checkpointed free list head pointer associated with the mispredicted branch frees the mappings of instructions after the branch in bulk (bulk-free)

Rename Map Table: The main problem is restoring the RMT to the point of the branch

Restoring the RMT

1. Copy AMT to RMT right away (the head's version)
2. Fast forward the RMT to the point of the branch
    1. Walk the AL from head to branch
    2. There maybe multiple mappings of a logical register between head and branch. We want the one closest to the branch. In the below, we care about restoring RMT to <r1,p8>



Restoring RMT

Recall that the active list contains current mappings, which are used to update the AMT (and free up prevoius register mappings stored in AMT)

When misprediction is detected:

- *Restore RMT (RAT) from AMT (RRAT) right away (flash or serial copy)*
- *Fast forward the RMT to the point of the branch in the active list, by walking the active list from head to branch and incrementally updating the RMT with the current mappings in the active list*

Contrast to the earlier approach with AMT

- Copy AMT into RMT when the branch reaches the head (lazy recovery)
- Lazy recovery has a serious drawback: All branches prior to the branch must retire before recovery kicks in (think a long-latency memory operation at head of AL)
- *The latency of lazy recovery is proportional to the time it takes to execute the instructions prior to the branch. The latency of eager recovery is proportional to the # instructions before the branch (and not on the time it takes to execute them)*

Rollback-Based Recovery (Scenario # 2)

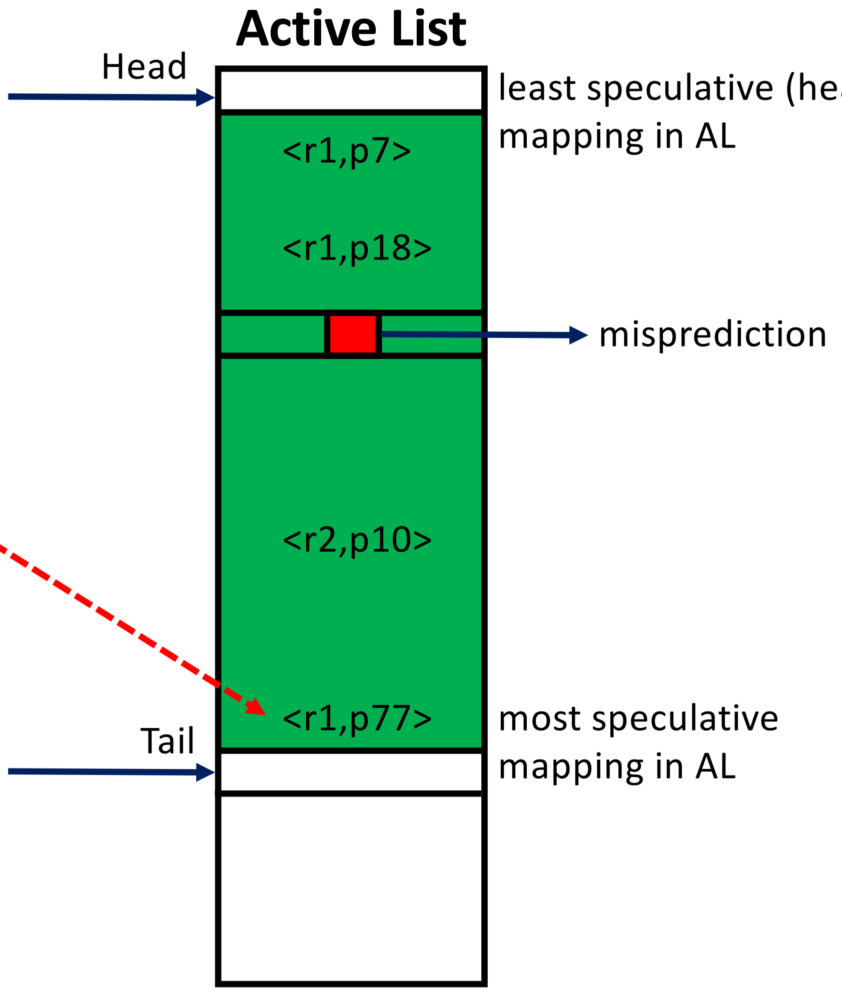
- Scenario is as follows
  - AMT in the backend
  - Active list contains the **current** logical→physical mappings
  - **Eager recovery:** As soon as branch misprediction is discovered, initiate recovery
    - **Note:** Multiple branches are a complication everywhere
  - Real-life example: Cyrix III, VIA Technologies

Early Misprediction Recovery (+AMT)

RMT (speculative)

Logical	Physical
r1	p77
r2	p10

most speculative mapping


Active List
Head
<r1,p7>
<r1,p18>
misprediction
<r2,p10>
<r1,p77>
Tail
least speculative (head) mapping in AL
most speculative mapping in AL

AMT (Head version)

current arch mapping

Logical	Physical
r1	p21
r2	p98

Early Misprediction Recovery (+AMT)

RMT (speculative)

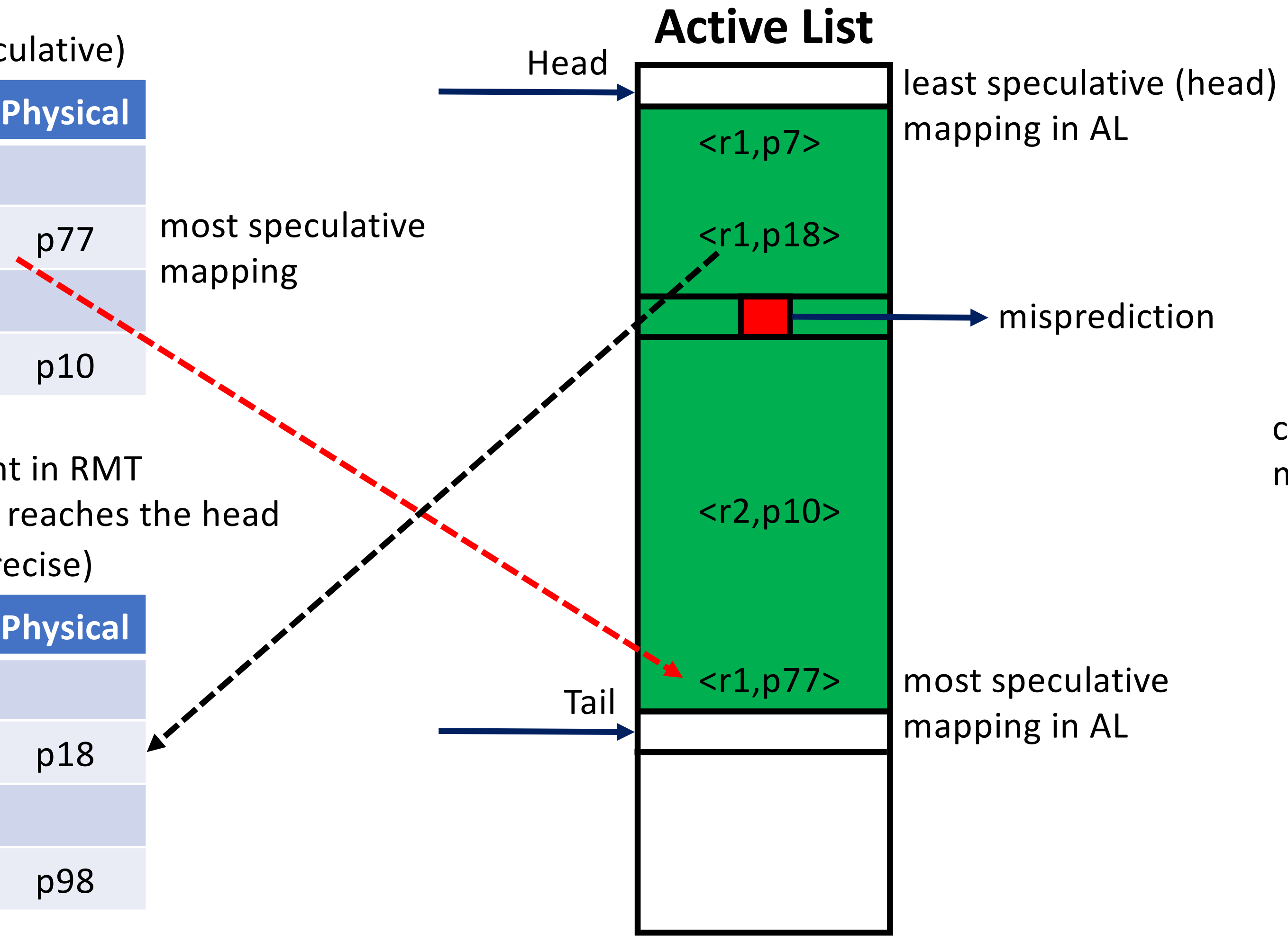
Logical	Physical
r1	p77
r2	p10

most speculative mapping

What we want in RMT when branch reaches the head

RMT (precise)

Logical	Physical
r1	p18
r2	p98



Active List
Head
least speculative (head) mapping in AL
<r1,p7>
<r1,p18>
misprediction
<r2,p10>
<r1,p77>
most speculative mapping in AL
Tail

AMT (Head version)

current arch mapping

Logical	Physical
r1	p21
r2	p98

Early Misprediction Recovery (+AMT)

RMT (speculative)

Logical	Physical
r1	p77
r2	p10

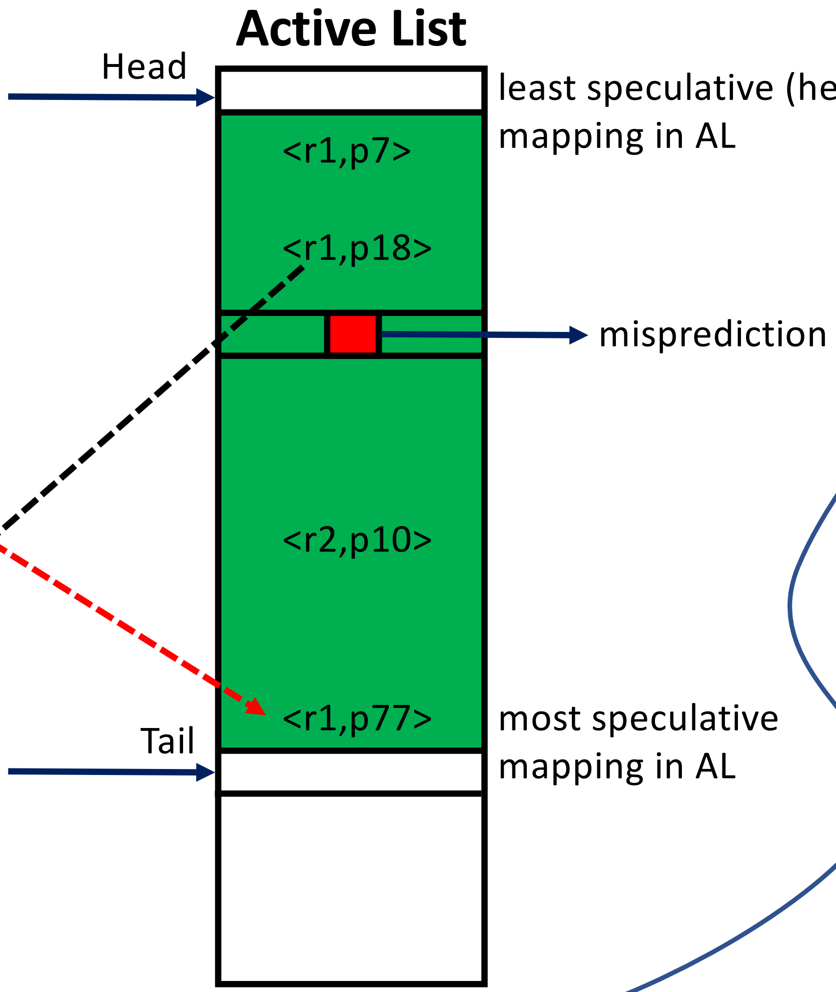
most speculative mapping

What we want in RMT when branch reaches the head

RMT (precise)

Logical	Physical
r1	p18
r2	p98

To Here



Active List
Head
least speculative (head) mapping in AL
<r1,p7>
<r1,p18>
misprediction
<r2,p10>
<r1,p77>
Tail
most speculative mapping in AL

From Here

AMT (Head version)

Logical	Physical
r1	p21
r2	p98

current arch mapping

Repairing RMT

- Copy AMT into RMT
- Fast-forward the RMT by walking the AL from Head to the mispredicted branch

Shadow Map Tables

- Checkpoint (create a copy) the RMT at every predicted branch
  - After the branch is renamed, the state of RMT reflects the renaming of all instructions up to and including the branch

When branch executes ("resolves")

- Misprediction
  - Restore RMT from mispredicted branch's shadow map table (SMT)
  - Reclaim (free) the mispredicted branch's SMT
  - Reclaims the shadow maps of all later branches for use by new predicted branches in program order
- Correct predicton
  - Reclaim the branch's shadow map table, for use by new predicted branches

Checkpoints vs. Rollback Recovery

- **Rollback-based recovery**
  - Slow, can’t undo all instructions in one cycle (only 4 to 8)
- **Checkpointing**
  - Take a snapshot of the RMT when the branch is renamed
  - Structures: Shadow maps, shadow registers, branch stack
  - Branch stack in MIPS R10K (4 entries)
  - HaL PM1 (SPARC64) had 16 shadow registers
  - ALPHA 21264 had 80 snapshopts (one for each ROB instruction)
  - Complications and opportunities
    - Multiple branches
    - Selective squash (branch masks in MIPS R10K)

Overall Branch Misprediction Recovery

Pipeline

- Instruction fetch unit
  - PC = Correct branch target
  - Repair the BHR if applicable
- Frontend stages: fetch, decode, rename, dispatch
  - Squash all instructions in the frontend stages since these are after the resolved branch (by definition)
- Backend stages: schedule (issue), register read, execute, writeback
  - *Selectively squash only those instructions in these stages that come after the mispredicted branch in program order (with eager recovery)*
  - Each instruction inherits a vector of unresolved branch identifiers in the rename stage, indicating which unresolved branches are before the instruction in program order

Overall Branch Misprediction Recovery

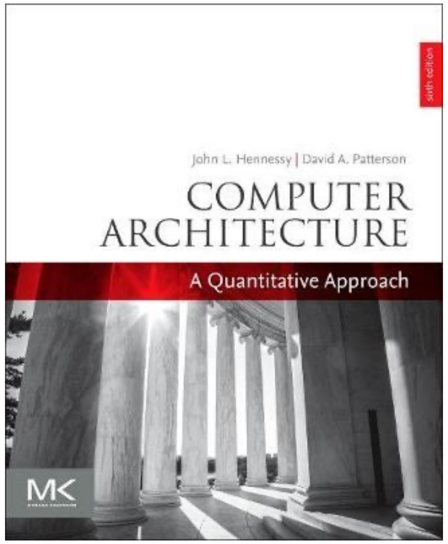
Active List
  - Set tail pointer to entry just after the mispredicted branch
- Free List
  - Lazy approaches (Head = Tail), Eager approaches (checkpoint)
- RMT
  - Copy from AMT or shadow maps, or copy plus walk the AL
- Shadow map tables
  - Reclaim the shadow maps of mispredicted branch and all later branches

Readings

Microarchitecture of HaL's CPU

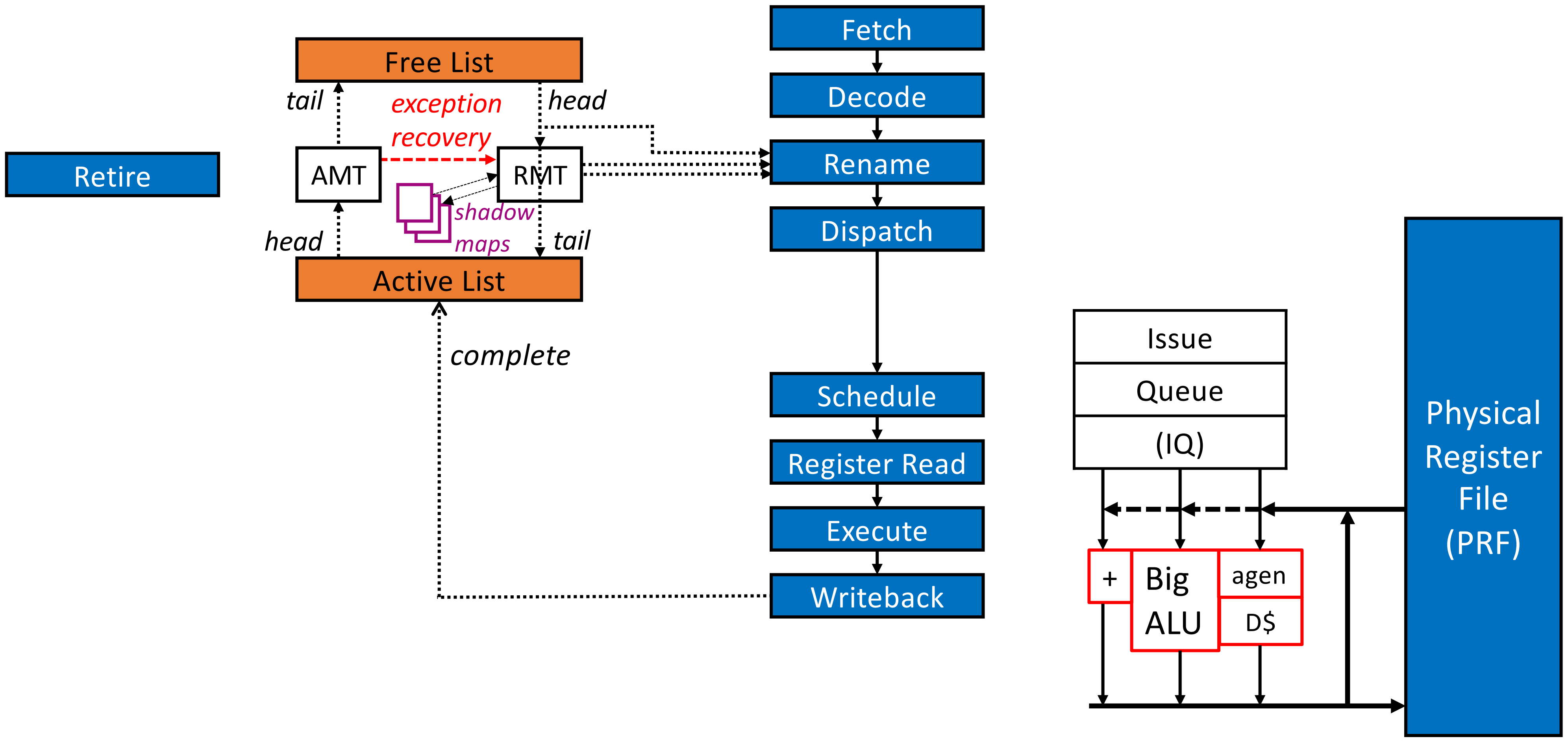
The Mips R10000 superscalar microprocessor

The Alpha 21264 microprocessor architecture



by John L Hennessy (Author), David A Patterson

Contemporary Superscalar Microarchitecture



Exercise (Sample Exam Q)

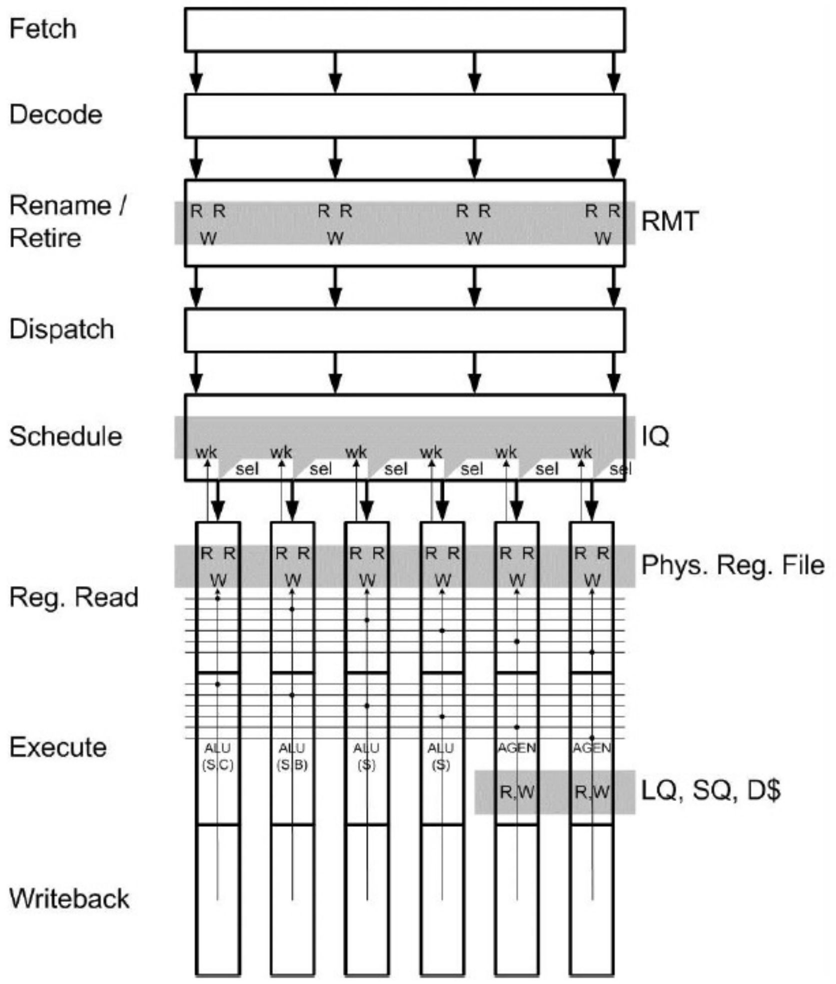
- Can we use no-data-capture in ARF+ROB without using tags, free list, and AMT?
  - What information do we need to maintain next to each entry in the ROB?
  - Think: Each register (value, rob_entry) is alive for a window of time
  - When the value is dead we can safely commit the value from the head of ROB?
  - How do we maintain lifetime information in ROB?

Superscalar Complexity

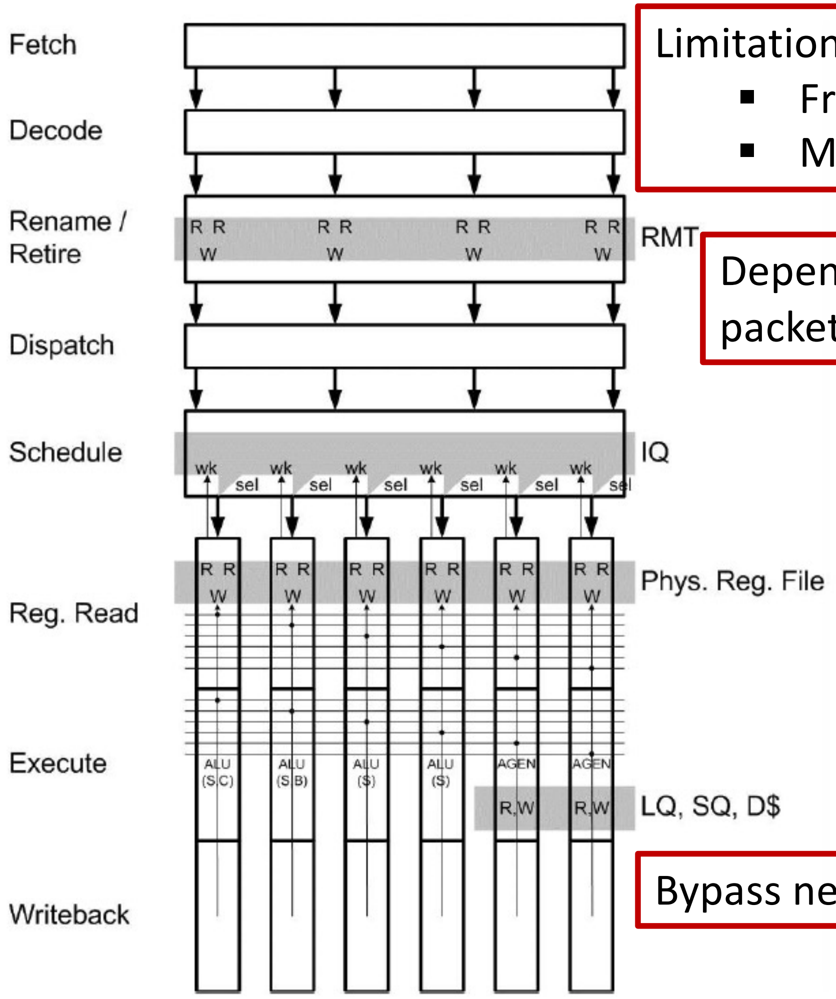
Superscalar and complexity

- Fetch, rename, dispatch, issue, and commit multiple instructions per cycle
- Use dynamic scheduling, renaming, and hardware speculation
- Goal: IPC>1 (ideal = issue width)
- Complexity increases with (issue) width
- Beyond 6-8 issue, the industry moved to multicores (?)
  - Complexity of very wide issue superscalar is not worth the increase in IPC. Better to exploit thread-level parallelism (TLP)
  - *So, after many decades of sustained performance increases, multicores(2005→) shifted the burden of perf. on software (how is that going?)*

4-Wide Issue superscalar



Superscalar Complexity



Limitations & complexity of fetching

- Frequent taken branches
- Multiple branch prediction

Dependences within the rename packet/bundle

Large # ports (RMT)

- Ports scale linearly with width of superscalar
- Sizes scale super-linearly with width to exploit ILP
- IQ, LQ, and SQ are CAMs (associative structures)
- Specialized logic (wake-up/select)

Bypass network complexity

Superscalar Complexity: Renaming

- Consider a 6-wide superscalar
  - All of the renaming in this instruction sequence (leftmost column) must happen in a single cycle
  - Increasing IPC (by issuing 6 instructions in one cycle) at the expense of increased cycle time likely leads to poor energy efficiency

Instr. #	Instruction	Physical Reg.	Renamed Inst.	RMT Change
1	add r1, r2, r3	p32	add p32, p2, p3	r1→p32
2	sub r1, r1, r2	p33	sub p33, p32, p2	r1→p33
3	add r2, r1, r2	p34	add p34, p33, p2	r2→p34
4	sub r1, r3, r2	p35	sub p35, p3, p34	r1→p35
5	add r1, r1, r2	p36	add p36, p35, p34	r1→p36
6	sub r1, r3, r1	p37	sub p37, p3, p36	r1→p37

What does the digital logic circuit looks like?