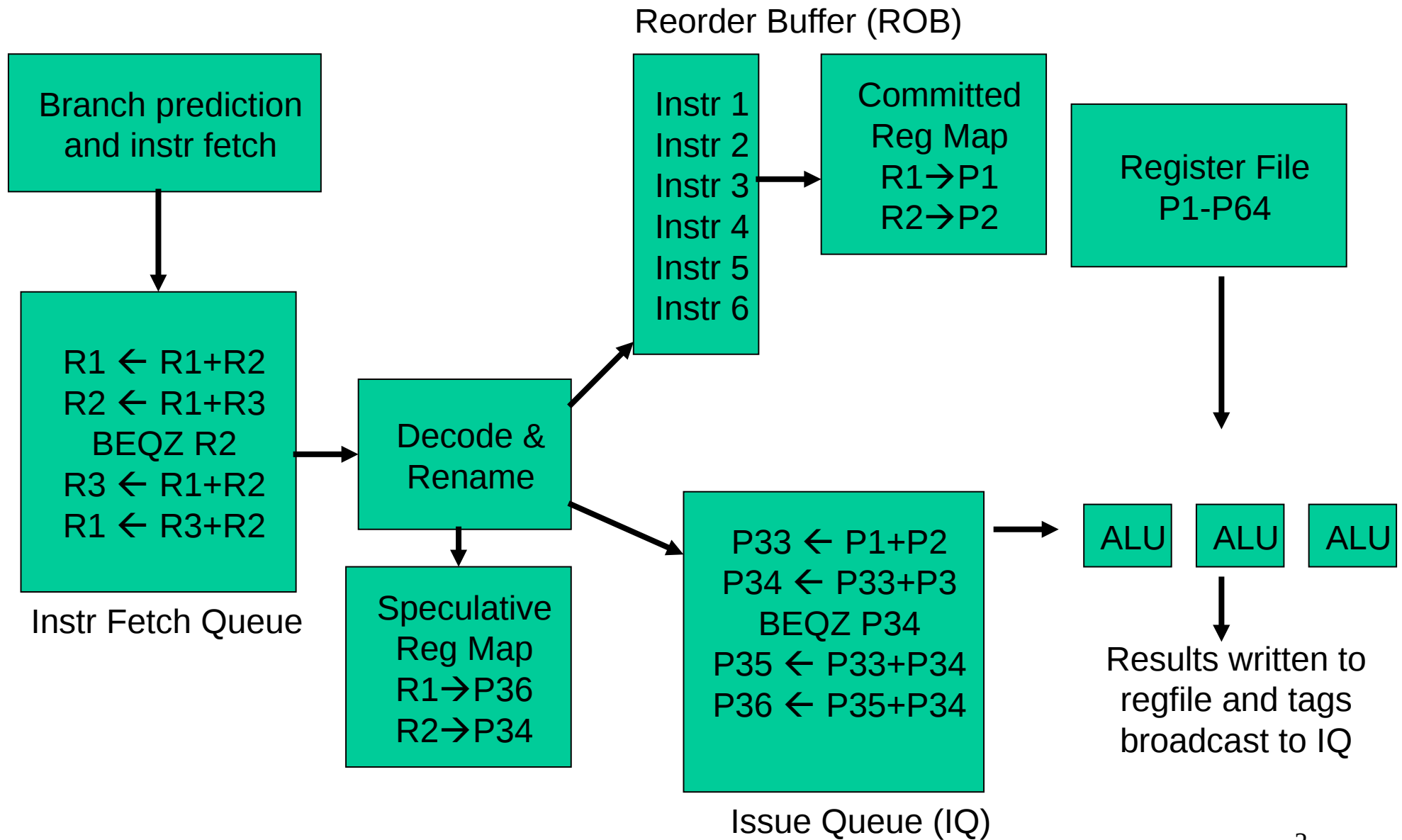


# 250P: Computer Systems Architecture

## Lecture 9: Out-of-order execution (continued)

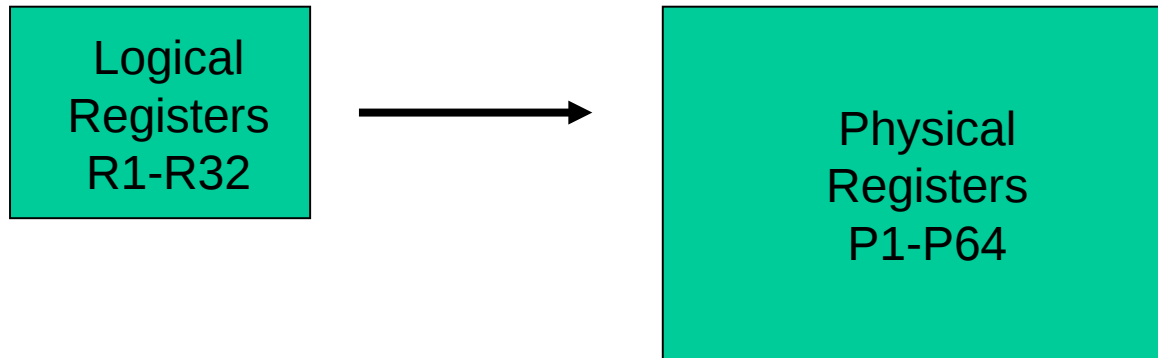
Anton Burtsev  
February, 2019

# The Alpha 21264 Out-of-Order Implementation

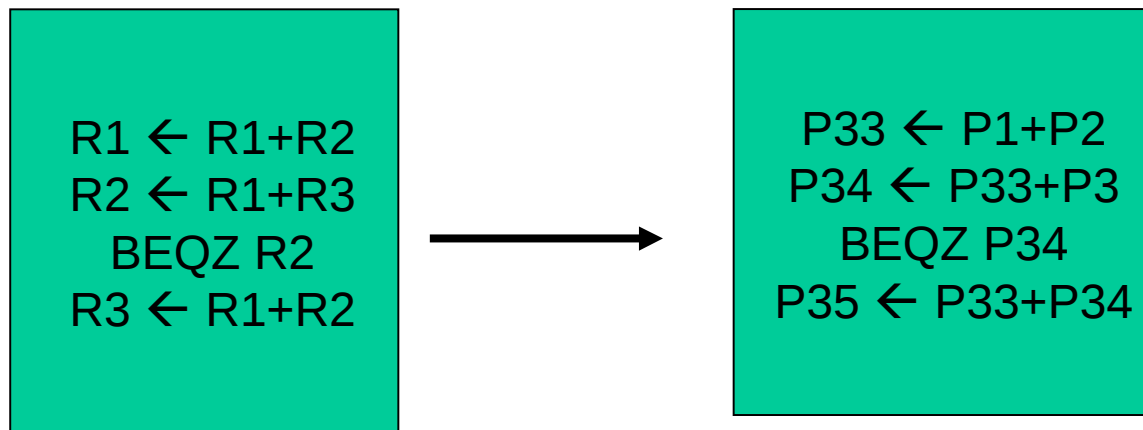


# Managing Register Names

Temporary values are stored in the register file and not the ROB



At the start, R1-R32 can be found in P1-P32  
Instructions stop entering the pipeline when P64 is assigned



What happens on commit?

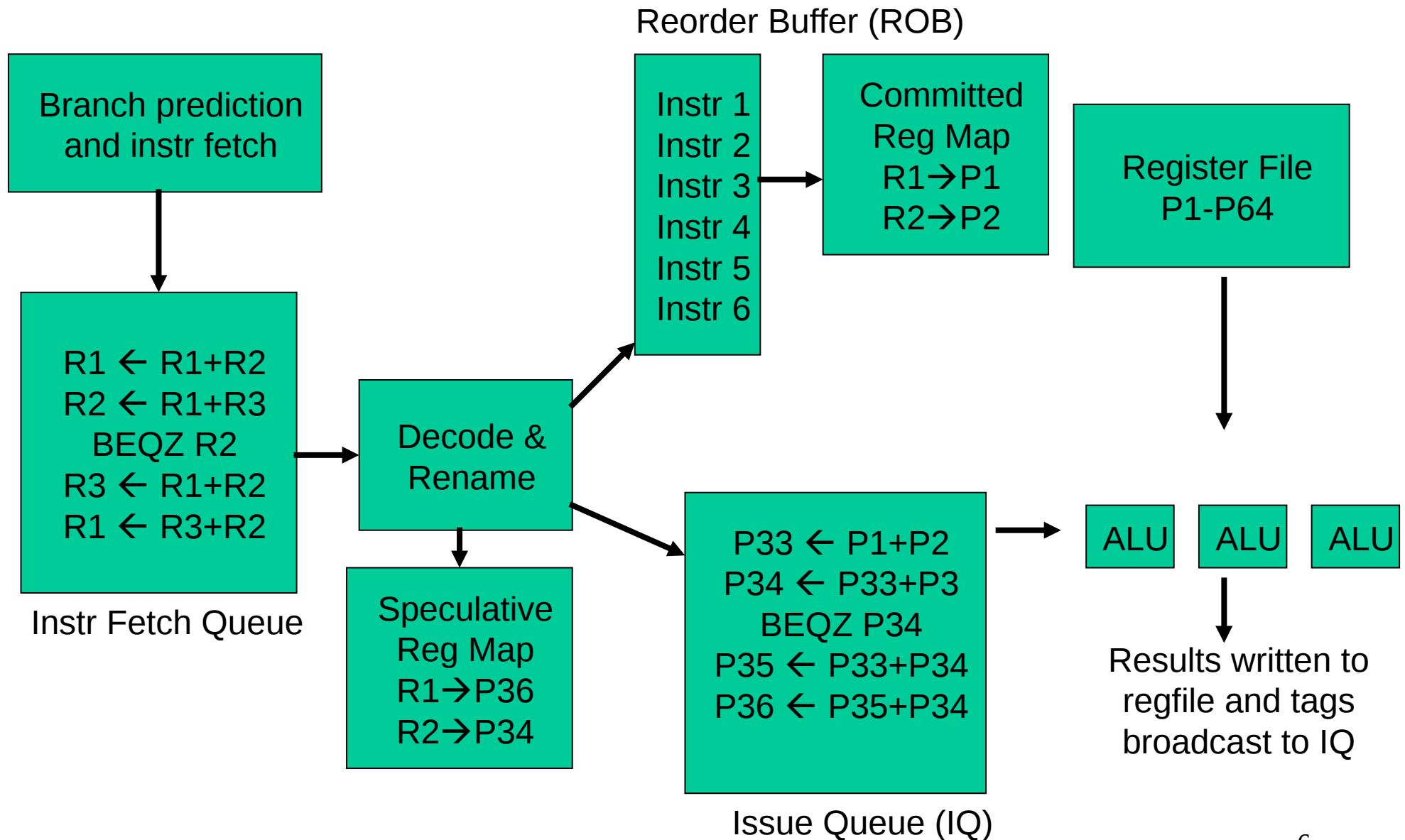
# The Commit Process

- On commit, no copy is required
- The register map table is updated – the “committed” value of R1 is now in P33 and not P1 – on an exception, P33 is copied to memory and not P1
- An instruction in the issue queue need not modify its input operand when the producer commits
- When instruction-1 commits, we no longer have any use for P1 – it is put in a free pool and a new instruction can now enter the pipeline → for every instr that commits, a new instr can enter the pipeline → number of in-flight instrs is a constant = number of extra (rename) registers

# Additional Details

- When does the decode stage stall? When we either run out of registers, or ROB entries, or issue queue entries
- Issue width: the number of instructions handled by each stage in a cycle. High issue width → high peak ILP
- Window size: the number of in-flight instructions in the pipeline. Large window size → high ILP
- No more WAR and WAW hazards because of rename registers – must only worry about RAW hazards

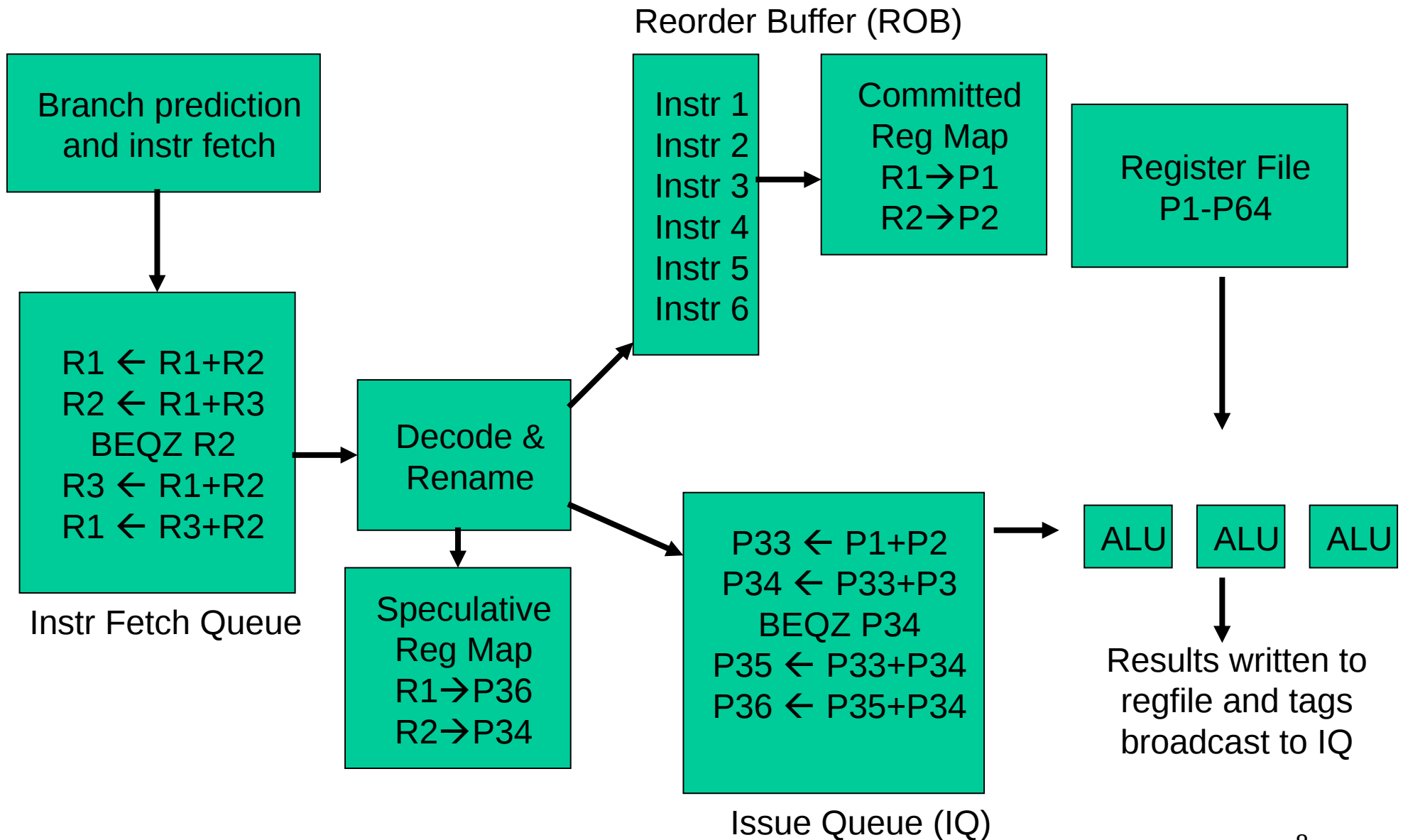
# The Alpha 21264 Out-of-Order Implementation



# Additional Details

- When does the decode stage stall? When we either run out of registers, or ROB entries, or issue queue entries
- Issue width: the number of instructions handled by each stage in a cycle. High issue width → high peak ILP
- Window size: the number of in-flight instructions in the pipeline. Large window size → high ILP
- No more WAR and WAW hazards because of rename registers – must only worry about RAW hazards

# The Alpha 21264 Out-of-Order Implementation





# Branch Mispredict Recovery

- On a branch mispredict, must roll back the processor state: throw away IFQ contents, ROB/IQ contents after branch
- Committed map table is correct and need not be fixed
- The speculative map table needs to go back to an earlier state
- To facilitate this spec-map-table rollback, it is checkpointed at every branch

# Waking Up a Dependent

- In an in-order pipeline, an instruction leaves the decode stage when it is known that the inputs can be correctly received, not when the inputs are computed
- Similarly, an instruction leaves the issue queue before its inputs are known, i.e., wakeup is speculative based on the expected latency of the producer instruction

# Out-of-Order Loads/Stores

Ld	R1 ← [R2]
Ld	R3 ← [R4]
St	R5 → [R6]
Ld	R7 ← [R8]
Ld	R9 ← [R10]

What if the issue queue also had load/store instructions?  
Can we continue executing instructions out-of-order?

# Memory Dependence Checking

Ld	0x abcdef
Ld	
St	
Ld	
Ld	0x abcdef
St	0x abcd00
Ld	0x abc000
Ld	0x abcd00

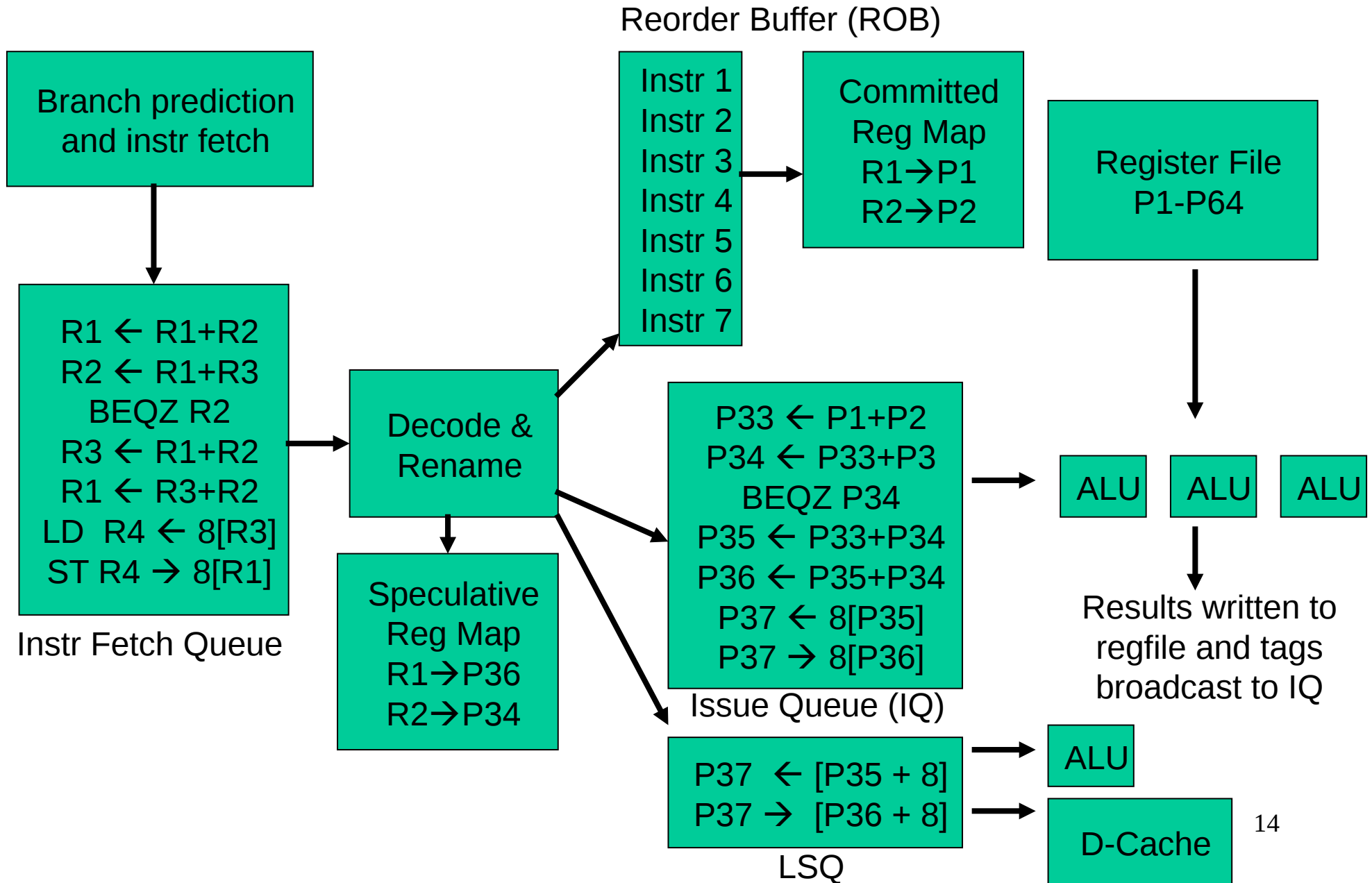
- The issue queue checks for register dependences and executes instructions as soon as registers are ready
- Loads/stores access memory as well – must check for RAW, WAW, and WAR hazards for memory as well
- Hence, first check for register dependences to compute effective addresses; then check for memory dependences

# Memory Dependence Checking

Ld	0x abcdef
Ld	
St	
Ld	
Ld	0x abcdef
St	0x abcd00
Ld	0x abc000
Ld	0x abcd00

- Load and store addresses are maintained in program order in the Load/Store Queue (LSQ)
- Loads can issue if they are guaranteed to not have true dependences with earlier stores
- Stores can issue only if we are ready to modify memory (can not recover if an earlier instr raises an exception)

# The Alpha 21264 Out-of-Order Implementation

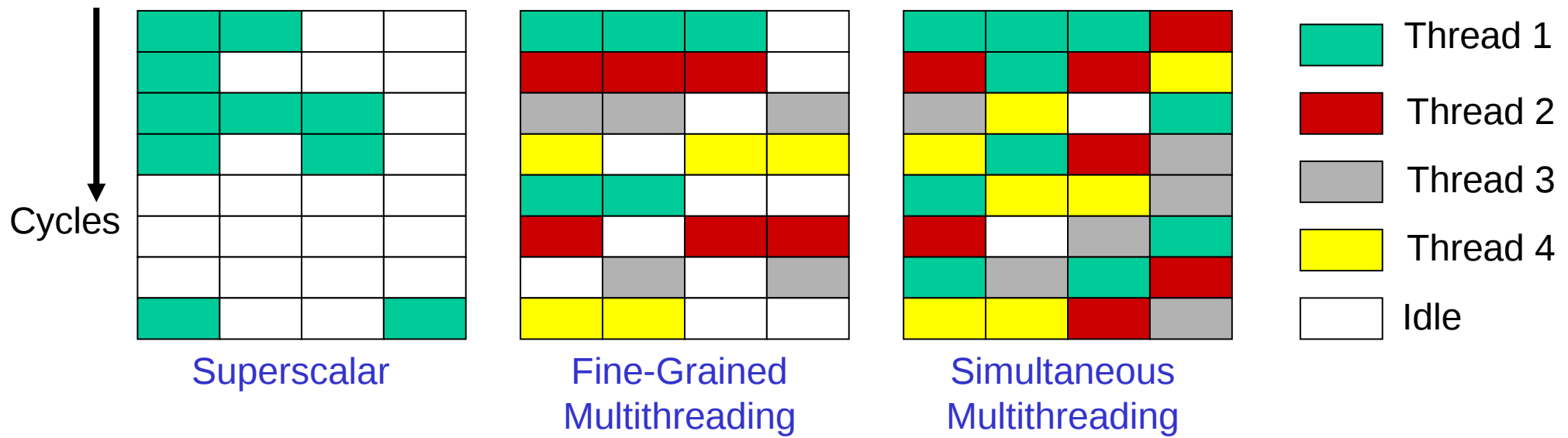


# Thread-Level Parallelism

- Motivation:
  - a single thread leaves a processor under-utilized for most of the time
  - by doubling processor area, single thread performance barely improves
- Strategies for thread-level parallelism:
  - multiple threads share the same large processor → reduces under-utilization, efficient resource allocation  
**Simultaneous Multi-Threading (SMT)**
  - each thread executes on its own mini processor → simple design, low interference between threads  
**Chip Multi-Processing (CMP) or multi-core**

# How are Resources Shared?

Each box represents an issue slot for a functional unit. Peak thruput is 4 IPC.



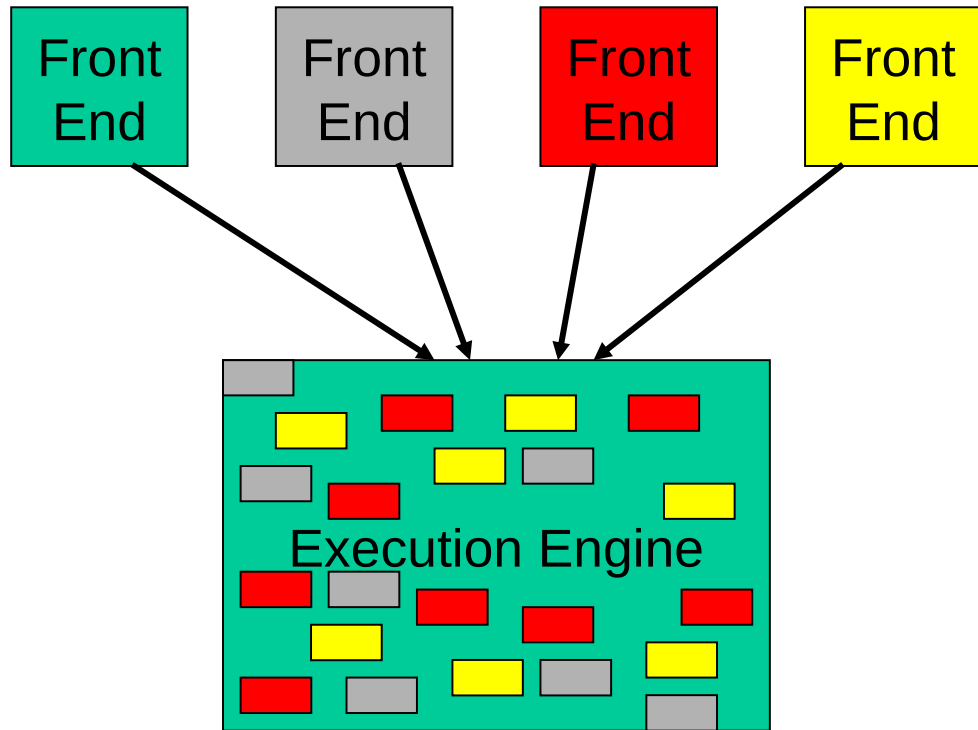
- Superscalar processor has high under-utilization – not enough work every cycle, especially when there is a cache miss
- Fine-grained multithreading can only issue instructions from a single thread in a cycle – can not find max work every cycle, but cache misses can be tolerated
- Simultaneous multithreading can issue instructions from any thread every cycle – has the highest probability of finding work for every issue slot



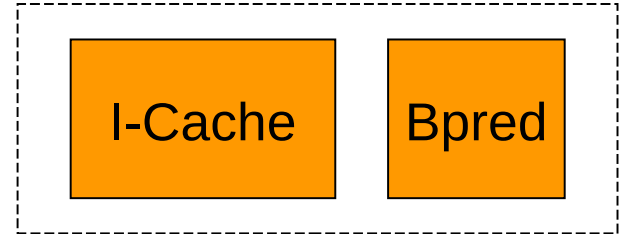
# What Resources are Shared?

- Multiple threads are simultaneously active (in other words, a new thread can start without a context switch)
- For correctness, each thread needs its own PC, IFQ, logical regs (and its own mappings from logical to phys regs)
- For performance, each thread could have its own ROB/LSQ (so that a stall in one thread does not stall commit in other threads), I-cache, branch predictor, D-cache, etc. (for low interference), although note that more sharing → better utilization of resources
- Each additional thread costs a PC, IFQ, rename tables, and ROB – cheap!

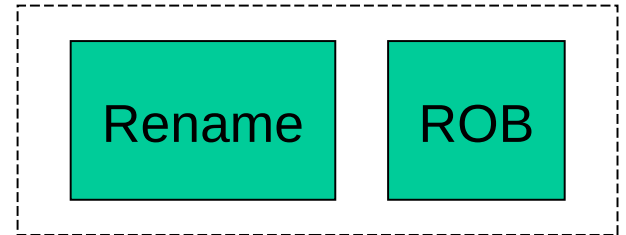
# Pipeline Structure



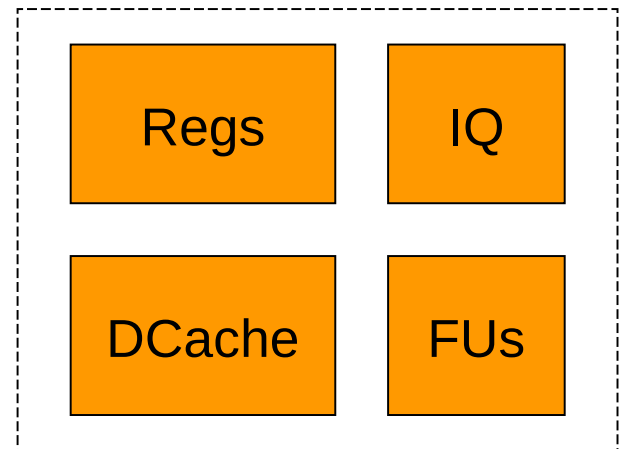
Private/  
Shared  
Front-end



Private  
Front-end



Shared  
Exec Engine



# Resource Sharing

Thread-1

$R1 \leftarrow R1 + R2$   
 $R3 \leftarrow R1 + R4$   
 $R5 \leftarrow R1 + R3$

Instr Fetch

Instr Fetch

$R2 \leftarrow R1 + R2$   
 $R5 \leftarrow R1 + R2$   
 $R3 \leftarrow R5 + R3$

Thread-2

$P65 \leftarrow P1 + P2$   
 $P66 \leftarrow P65 + P4$   
 $P67 \leftarrow P65 + P66$

Instr Rename

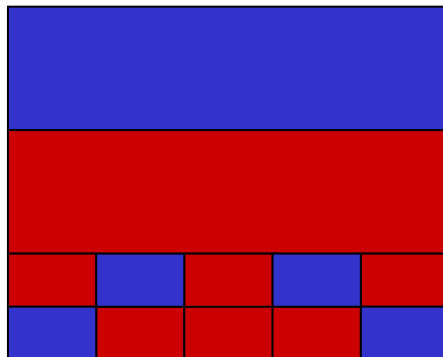
Instr Rename

$P76 \leftarrow P33 + P34$   
 $P77 \leftarrow P33 + P76$   
 $P78 \leftarrow P77 + P35$

Issue Queue

$P65 \leftarrow P1 + P2$   
 $P66 \leftarrow P65 + P4$   
 $P67 \leftarrow P65 + P66$   
 $P76 \leftarrow P33 + P34$   
 $P77 \leftarrow P33 + P76$   
 $P78 \leftarrow P77 + P35$

Register File



# Performance Implications of SMT

- Single thread performance is likely to go down (caches, branch predictors, registers, etc. are shared) – this effect can be mitigated by trying to prioritize one thread
- While fetching instructions, thread priority can dramatically influence total throughput – a widely accepted heuristic (ICOUNT): fetch such that each thread has an equal share of processor resources
- With eight threads in a processor with many resources, SMT yields throughput improvements of roughly 2-4

# Pentium4 Hyper-Threading

- Two threads – the Linux operating system operates as if it is executing on a two-processor system
- When there is only one available thread, it behaves like a regular single-threaded superscalar processor
- Statically divided resources: ROB, LSQ, issueq -- a slow thread will not cripple thruput (might not scale)
- Dynamically shared: trace cache and decode (fine-grained multi-threaded, round-robin), FUs, data cache, bpred

# Multi-Programmed Speedup

Benchmark	Best Speedup	Worst Speedup	Avg Speedup
gzip	1.48	1.14	1.24
vpr	1.43	1.04	1.17
gcc	1.44	1.00	1.11
mcf	1.57	1.01	1.21
crafty	1.40	0.99	1.17
parser	1.44	1.09	1.18
eon	1.42	1.07	1.25
perlbmk	1.40	1.07	1.20
gap	1.43	1.17	1.25
vortex	1.41	1.01	1.13
bzip2	1.47	1.15	1.24
twolf	1.48	1.02	1.16
wupwise	1.33	1.12	1.24
swim	1.58	0.90	1.13
mgrid	1.28	0.94	1.10
applu	1.37	1.02	1.16
mesa	1.39	1.11	1.22
galgel	1.47	1.05	1.25
art	1.55	0.90	1.13
equake	1.48	1.02	1.21
facerec	1.39	1.16	1.25
ammp	1.40	1.09	1.21
lucas	1.36	0.97	1.13
fma3d	1.34	1.13	1.20
sixtrack	1.58	1.28	1.42
apsi	1.40	1.14	1.23
Overall	1.58	0.90	1.20

- sixtrack and eon do not degrade their partners (small working sets?)
- swim and art degrade their partners (cache contention?)
- Best combination: swim & sixtrack  
worst combination: swim & art
- Static partitioning ensures low interference – worst slowdown is 0.9

## Problem 2

---

- Show the renamed version of the following code:  
Assume that you have 36 physical registers and 32 architected registers

R1  $\leftarrow$  R2+R3

R3  $\leftarrow$  R4+R5

BEQZ R1

R1  $\leftarrow$  R1 + R3

R1  $\leftarrow$  R1 + R3

R3  $\leftarrow$  R1 + R3

R4  $\leftarrow$  R3 + R1

## Problem 2

---

- Show the renamed version of the following code:  
Assume that you have 36 physical registers and 32 architected registers

R1  $\leftarrow$  R2+R3

R3  $\leftarrow$  R4+R5

BEQZ R1

R1  $\leftarrow$  R1 + R3

R1  $\leftarrow$  R1 + R3

R3  $\leftarrow$  R1 + R3

R4  $\leftarrow$  R3 + R1

P33  $\leftarrow$  P2+P3

P34  $\leftarrow$  P4+P5

BEQZ P33

P35  $\leftarrow$  P33+P34

P36  $\leftarrow$  P35+P34

P1  $\leftarrow$  P36+P34

P3  $\leftarrow$  P1+P36



# Problem 3

---

- Show the renamed version of the following code:  
Assume that you have 36 physical registers and 32 architected registers. When does each instr leave the IQ?

R1  $\leftarrow$  R2+R3

R1  $\leftarrow$  R1+R5

BEQZ R1

R1  $\leftarrow$  R4 + R5

R4  $\leftarrow$  R1 + R7

R1  $\leftarrow$  R6 + R8

R4  $\leftarrow$  R3 + R1

R1  $\leftarrow$  R5 + R9

## Problem 3

---

- Show the renamed version of the following code:  
Assume that you have 36 physical registers and 32 architected registers. When does each instr leave the IQ?

R1  $\leftarrow$  R2+R3

R1  $\leftarrow$  R1+R5

BEQZ R1

R1  $\leftarrow$  R4 + R5

R4  $\leftarrow$  R1 + R7

R1  $\leftarrow$  R6 + R8

R4  $\leftarrow$  R3 + R1

R1  $\leftarrow$  R5 + R9

P33  $\leftarrow$  P2+P3

P34  $\leftarrow$  P33+P5

BEQZ P34

P35  $\leftarrow$  P4+P5

P36  $\leftarrow$  P35+P7

P1  $\leftarrow$  P6+P8

P33  $\leftarrow$  P3+P1

P34  $\leftarrow$  P5+P9

# Problem 3

---

- Show the renamed version of the following code:  
Assume that you have 36 physical registers and 32 architected registers. When does each instr leave the IQ?

R1 $\leftarrow$ R2+R3	P33 $\leftarrow$ P2+P3	cycle i
R1 $\leftarrow$ R1+R5	P34 $\leftarrow$ P33+P5	i+1
BEQZ R1	BEQZ P34	i+2
R1 $\leftarrow$ R4 + R5	P35 $\leftarrow$ P4+P5	i
R4 $\leftarrow$ R1 + R7	P36 $\leftarrow$ P35+P7	i+1
R1 $\leftarrow$ R6 + R8	P1 $\leftarrow$ P6+P8	j
R4 $\leftarrow$ R3 + R1	P33 $\leftarrow$ P3+P1	j+1
R1 $\leftarrow$ R5 + R9	P34 $\leftarrow$ P5+P9	j+2

Width is assumed to be 4.

j depends on the #stages between issue and commit.

Thank you!