

Lecture: Out-of-order Processors

- Topics: more ooo design details, timing, load-store queue

Problem 1

- 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 0

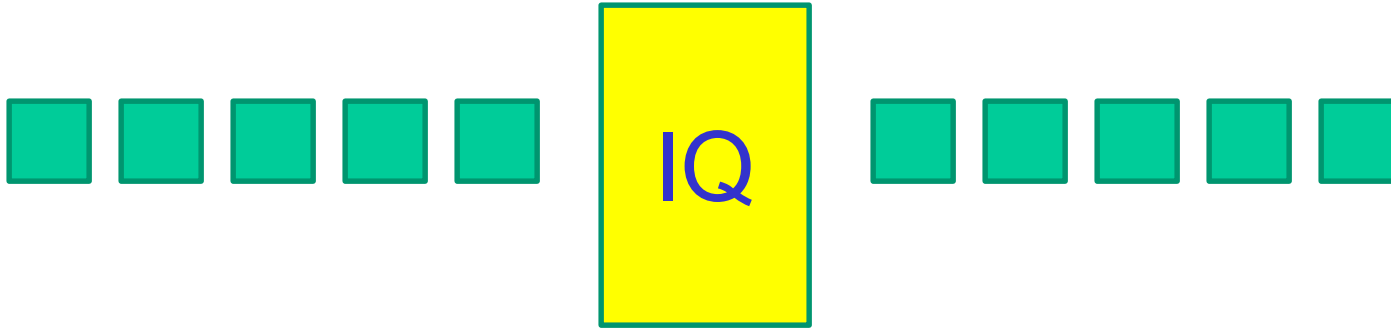
- Show the renamed version of the following code:
Assume that you have 36 physical registers and 32 architected registers. When does each instr leave the IQ?

$R1 \leftarrow R2+R3$	$P33 \leftarrow P2+P3$	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.

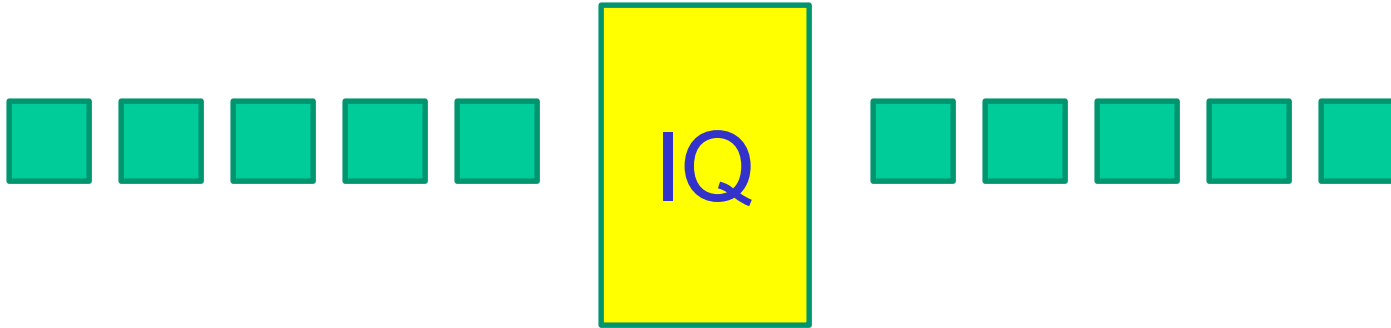
j depends on the #stages between issue and commit.

OOO Example



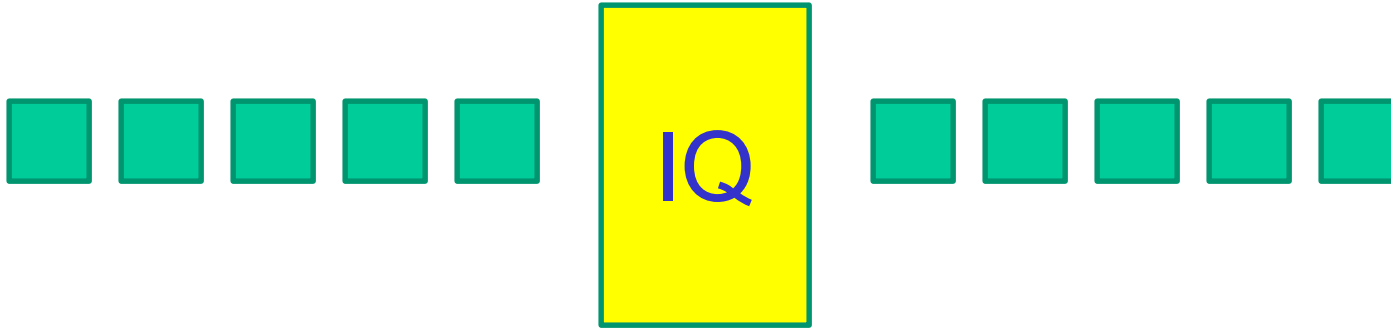
- Assume there are 36 physical registers and 32 logical registers, and width is 4
- Estimate the issue time, completion time, and commit time for the sample code

Assumptions



- Perfect branch prediction, instruction fetch, caches
- ADD → dep has no stall; LD → dep has one stall
- An instr is placed in the IQ at the end of its 5th stage, an instr takes 5 more stages after leaving the IQ (ld/st instrs take 6 more stages after leaving the IQ)

OOO Example

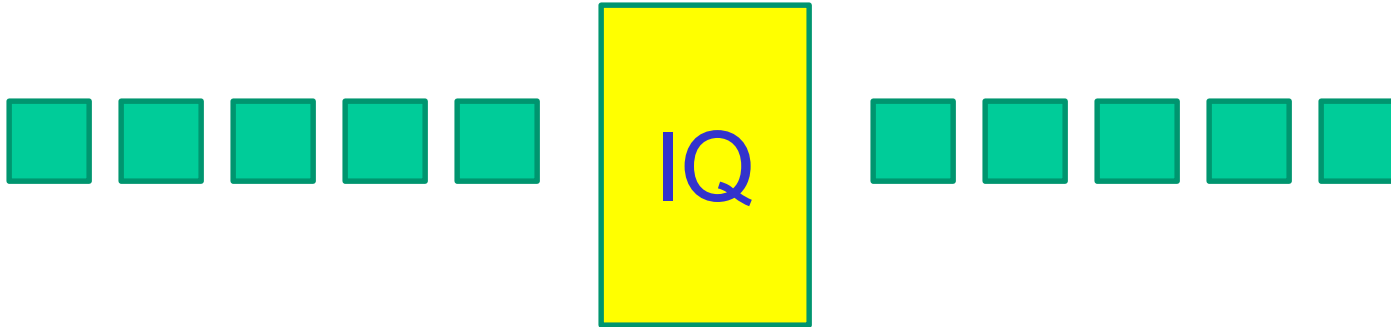


Original code

```
ADD R1, R2, R3
LD R2, 8(R1)
ADD R2, R2, 8
ST R1, (R3)
SUB R1, R1, R5
LD R1, 8(R2)
ADD R1, R1, R2
```

Renamed code

OOO Example



Original code

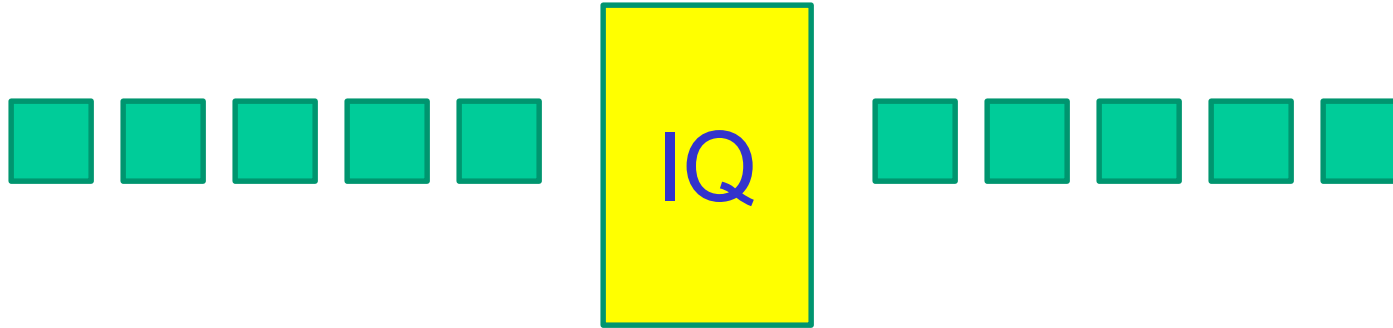
```
ADD R1, R2, R3
LD R2, 8(R1)
ADD R2, R2, 8
ST R1, (R3)
SUB R1, R1, R5
LD R1, 8(R2)
ADD R1, R1, R2
```

Renamed code

```
ADD P33, P2, P3
LD P34, 8(P33)
ADD P35, P34, 8
ST P33, (P3)
SUB P36, P33, P5
```

Must wait

OOO Example



Original code

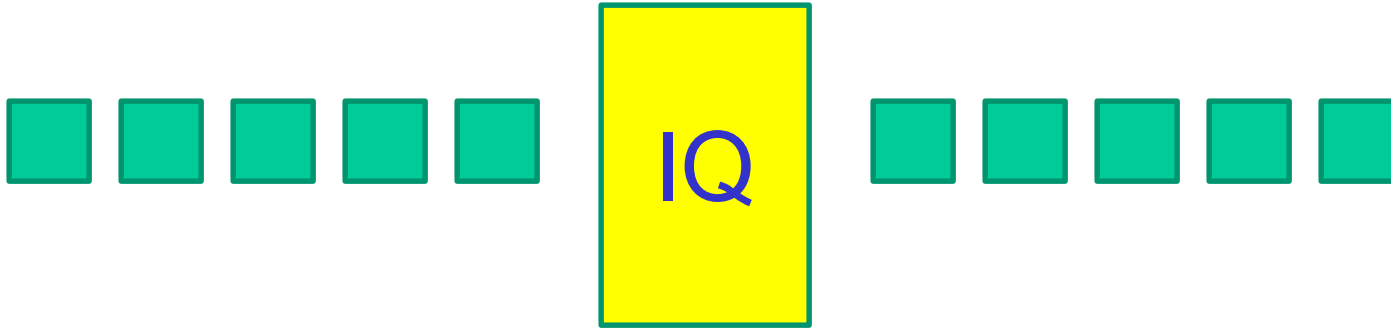
```
ADD R1, R2, R3
LD R2, 8(R1)
ADD R2, R2, 8
ST R1, (R3)
SUB R1, R1, R5
LD R1, 8(R2)
ADD R1, R1, R2
```

Renamed code

```
ADD P33, P2, P3
LD P34, 8(P33)
ADD P35, P34, 8
ST P33, (P3)
SUB P36, P33, P5
```

InQ Iss Comp Comm

OOO Example



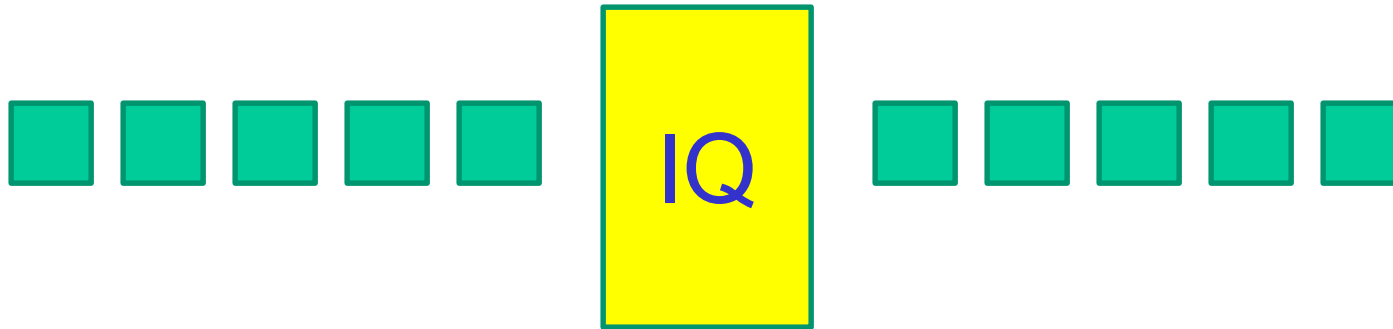
Original code

Renamed code

InQ Iss Comp Comm

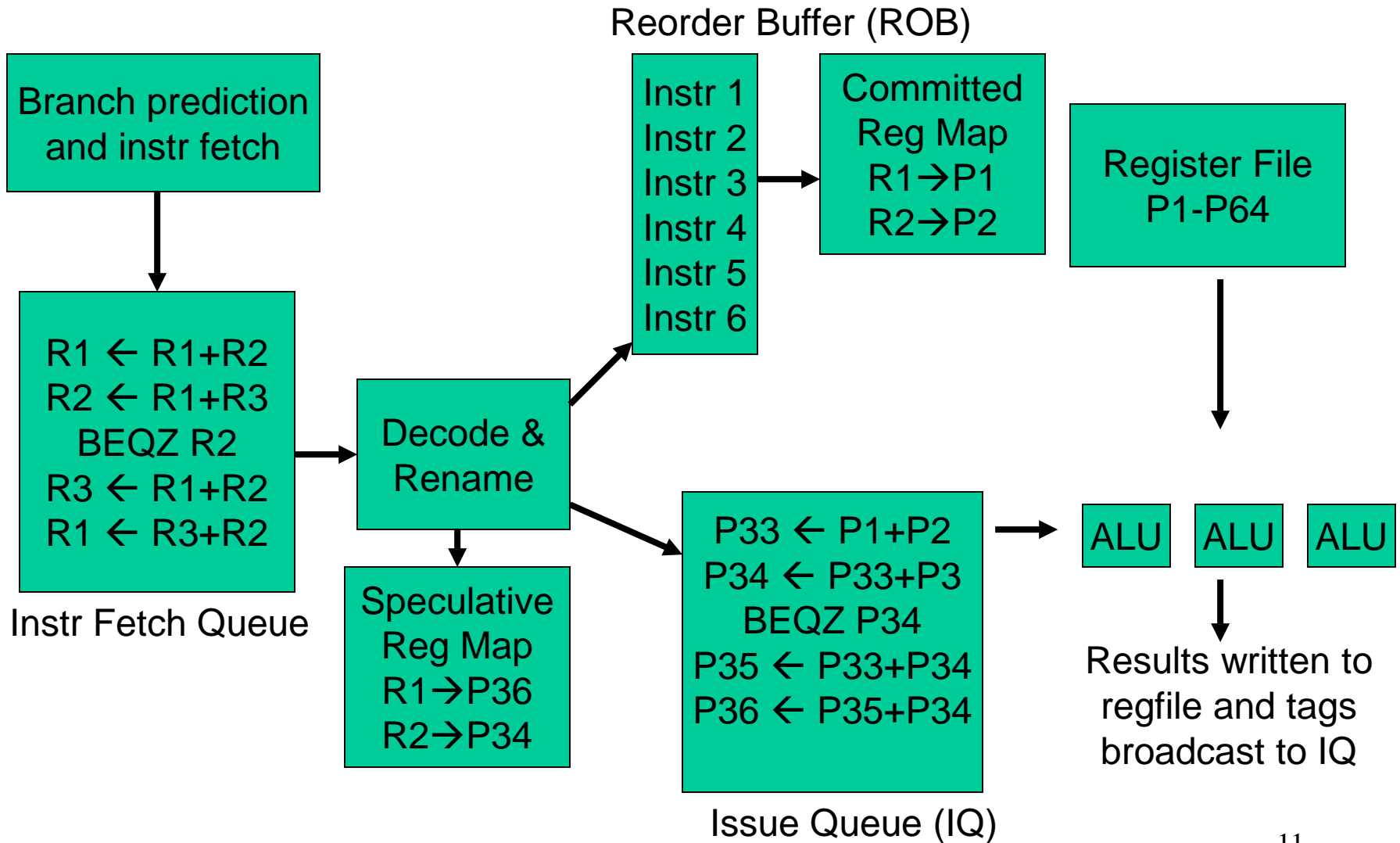
ADD	R1, R2, R3	ADD	P33, P2, P3	i	i+1	i+6	i+6
LD	R2, 8(R1)	LD	P34, 8(P33)	i	i+2	i+8	i+8
ADD	R2, R2, 8	ADD	P35, P34, 8	i	i+4	i+9	i+9
ST	R1, (R3)	ST	P33, (P3)	i	i+2	i+8	i+9
SUB	R1, R1, R5	SUB	P36, P33, P5	i+1	i+2	i+7	i+9
LD	R1, 8(R2)						
ADD	R1, R1, R2						

OOO Example



Original code	Renamed code	InQ	Iss	Comp	Comm
ADD R1, R2, R3	ADD P33, P2, P3	i	i+1	i+6	i+6
LD R2, 8(R1)	LD P34, 8(P33)	i	i+2	i+8	i+8
ADD R2, R2, 8	ADD P35, P34, 8	i	i+4	i+9	i+9
ST R1, (R3)	ST P33, (P3)	i	i+2	i+8	i+9
SUB R1, R1, R5	SUB P36, P33, P5	i+1	i+2	i+7	i+9
LD R1, 8(R2)	LD P1, 8(P35)	i+7	i+8	i+14	i+14
ADD R1, R1, R2	ADD P2, P1, P35	i+9	i+10	i+15	i+15

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

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

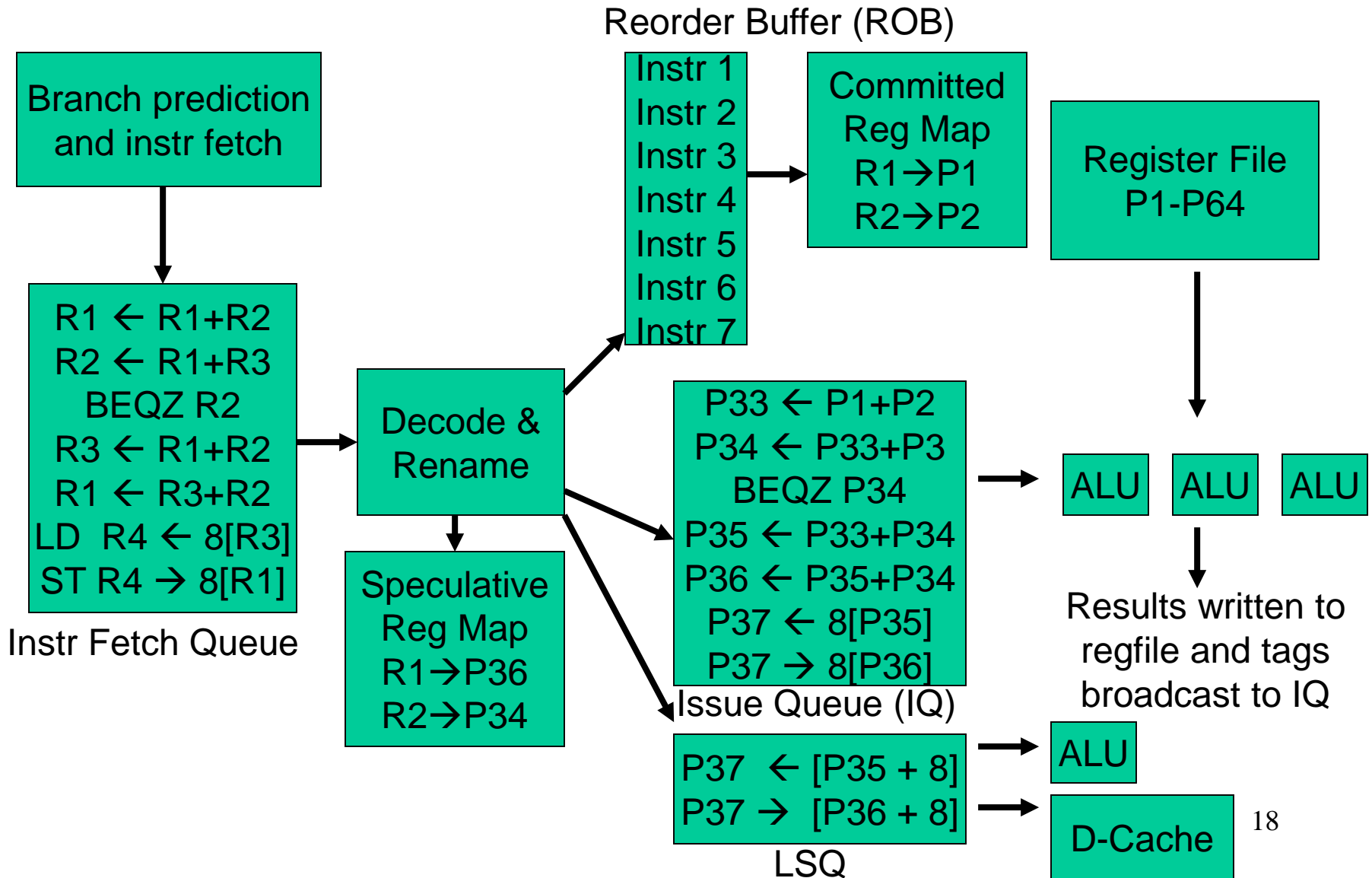
- The issue queue checks for register dependences and executes instructions as soon as registers are ready
- Loads/stores access memory as well – must check for RAW, WAW, and WAR hazards for memory as well
- Hence, first check for register dependences to compute effective addresses; then check for memory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

The Alpha 21264 Out-of-Order Implementation



Problem 2

- Consider the following LSQ and when operands are available. Estimate when the address calculation and memory accesses happen for each ld/st. Assume no memory dependence prediction.

		Ad. Op	St. Op	Ad.Val	Ad.Cal	Mem.Acc
LD	R1 ← [R2]	3		abcd		
LD	R3 ← [R4]	6		adde		
ST	R5 → [R6]	4	7	abba		
LD	R7 ← [R8]	2		abce		
ST	R9 → [R10]	8	3	abba		
LD	R11 ← [R12]	1		abba		

Problem 2

- Consider the following LSQ and when operands are available. Estimate when the address calculation and memory accesses happen for each ld/st. Assume no memory dependence prediction.

		Ad. Op	St. Op	Ad.Val	Ad.Cal	Mem.Acc
LD	R1 ← [R2]	3		abcd	4	5
LD	R3 ← [R4]	6		adde	7	8
ST	R5 → [R6]	4	7	abba	5	commit
LD	R7 ← [R8]	2		abce	3	6
ST	R9 → [R10]	8	3	abba	9	commit
LD	R11 ← [R12]	1		abba	2	10

Problem 3

- Consider the following LSQ and when operands are available. Estimate when the address calculation and memory accesses happen for each ld/st. Assume no memory dependence prediction.

		Ad. Op	St. Op	Ad.Val	Ad.Cal	Mem.Acc
LD	R1 ← [R2]	3		abcd		
LD	R3 ← [R4]	6		adde		
ST	R5 → [R6]	5	7	abba		
LD	R7 ← [R8]	2		abce		
ST	R9 → [R10]	1	4	abba		
LD	R11 ← [R12]	2		abba		

Problem 3

- Consider the following LSQ and when operands are available. Estimate when the address calculation and memory accesses happen for each ld/st. Assume no memory dependence prediction.

		Ad. Op	St. Op	Ad.Val	Ad.Cal	Mem.Acc
LD	R1 ← [R2]	3		abcd	4	5
LD	R3 ← [R4]	6		adde	7	8
ST	R5 → [R6]	5	7	abba	6	commit
LD	R7 ← [R8]	2		abce	3	7
ST	R9 → [R10]	1	4	abba	2	commit
LD	R11 ← [R12]	2		abba	3	5

Problem 4

- Consider the following LSQ and when operands are available. Estimate when the address calculation and memory accesses happen for each ld/st. Assume memory dependence prediction.

		Ad. Op	St. Op	Ad.Val	Ad.Cal	Mem.Acc
LD	R1 ← [R2]	3		abcd		
LD	R3 ← [R4]	6		adde		
ST	R5 → [R6]	4	7	abba		
LD	R7 ← [R8]	2		abce		
ST	R9 → [R10]	8	3	abba		
LD	R11 ← [R12]	1		abba		

Problem 4

- Consider the following LSQ and when operands are available. Estimate when the address calculation and memory accesses happen for each ld/st. Assume memory dependence prediction.

		Ad. Op	St. Op	Ad.Val	Ad.Cal	Mem.Acc
LD	R1 ← [R2]	3		abcd	4	5
LD	R3 ← [R4]	6		adde	7	8
ST	R5 → [R6]	4	7	abba	5	commit
LD	R7 ← [R8]	2		abce	3	4
ST	R9 → [R10]	8	3	abba	9	commit
LD	R11 ← [R12]	1		abba	2	3/10

Title

- Bullet