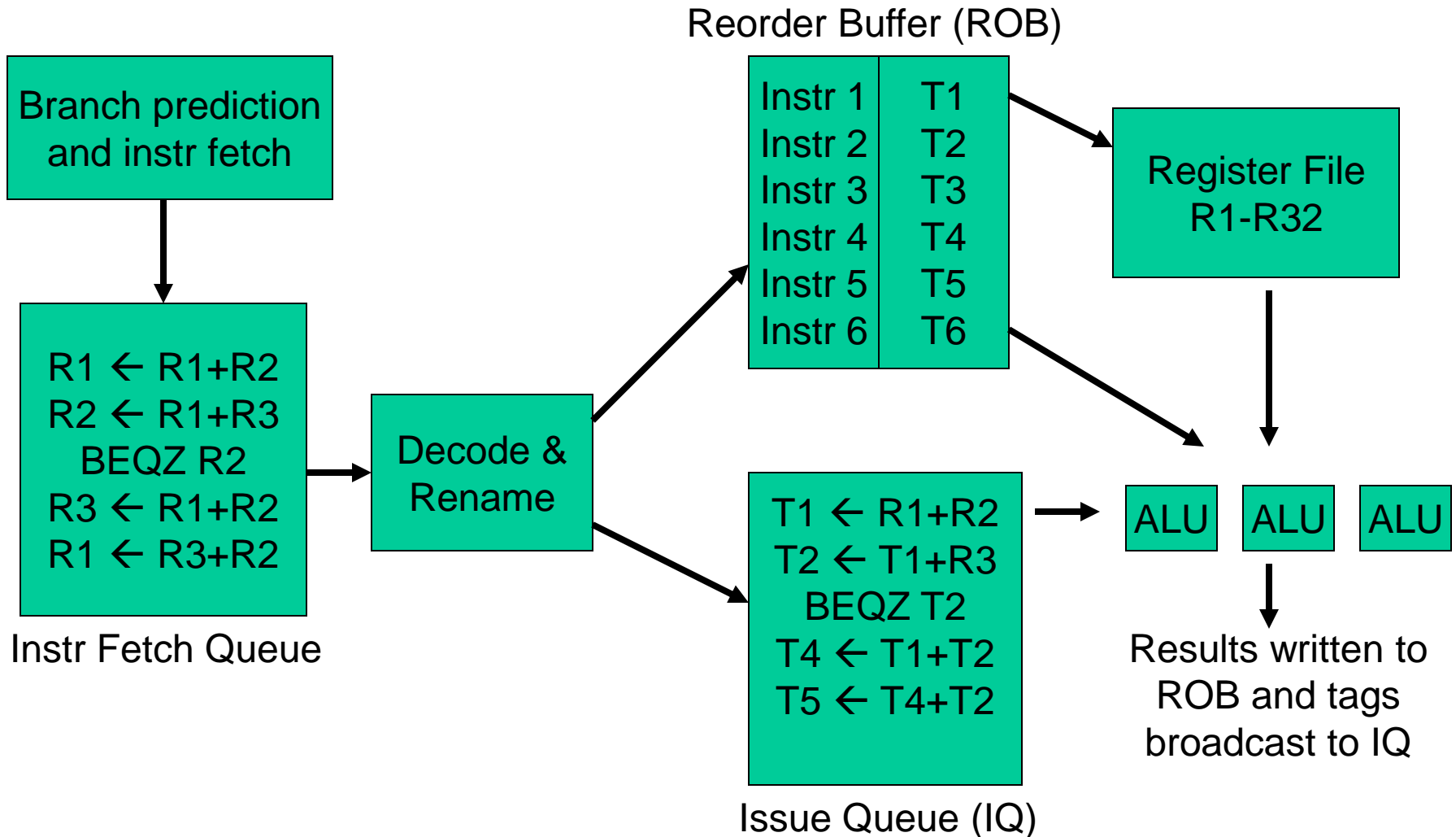


# Lecture 8: Dynamic ILP

---

- Topics: out-of-order processors  
(See class notes)
- HW3 is posted, due on Tuesday

# An Out-of-Order Processor Implementation



# Design Details - I

---

- Instructions enter the pipeline in order
- No need for branch delay slots if prediction happens in time
- Instructions leave the pipeline in order – all instructions that enter also get placed in the ROB – the process of an instruction leaving the ROB (in order) is called commit – an instruction commits only if it and all instructions before it have completed successfully (without an exception)
- To preserve precise exceptions, a result is written into the register file only when the instruction commits – until then, the result is saved in a temporary register in the ROB

# Design Details - II

---

- Instructions get renamed and placed in the issue queue – some operands are available (T1-T6; R1-R32), while others are being produced by instructions in flight (T1-T6)
- As instructions finish, they write results into the ROB (T1-T6) and broadcast the operand tag (T1-T6) to the issue queue – instructions now know if their operands are ready
- When a ready instruction issues, it reads its operands from T1-T6 and R1-R32 and executes (out-of-order execution)
- Can you have WAW or WAR hazards? By using more names (T1-T6), name dependences can be avoided

# Design Details - III

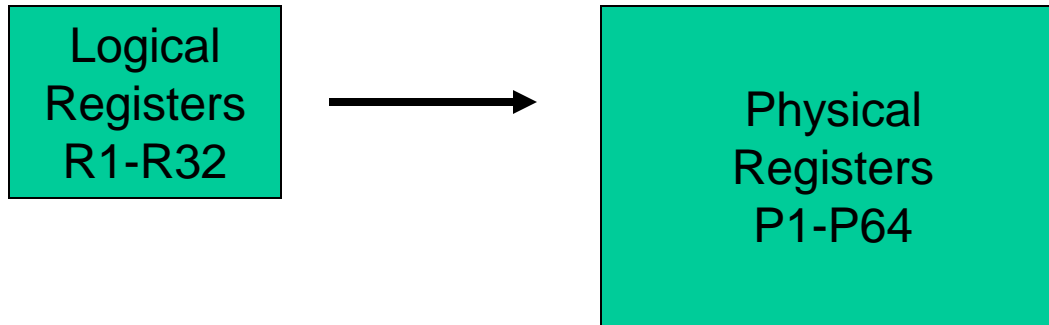
---

- If instr-3 raises an exception, wait until it reaches the top of the ROB – at this point, R1-R32 contain results for all instructions up to instr-3 – save registers, save PC of instr-3, and service the exception
- If branch is a mispredict, flush all instructions after the branch and start on the correct path – mispredicted instrs will not have updated registers (the branch cannot commit until it has completed and the flush happens as soon as the branch completes)
- Potential problems: ?

# Managing Register Names

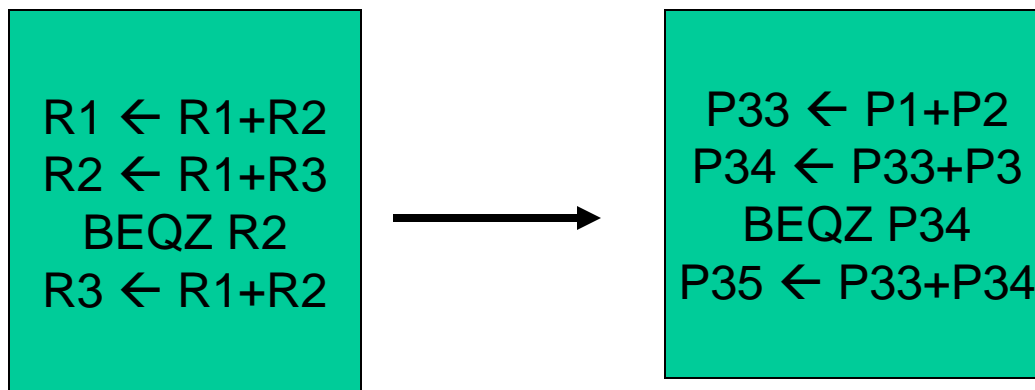
---

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



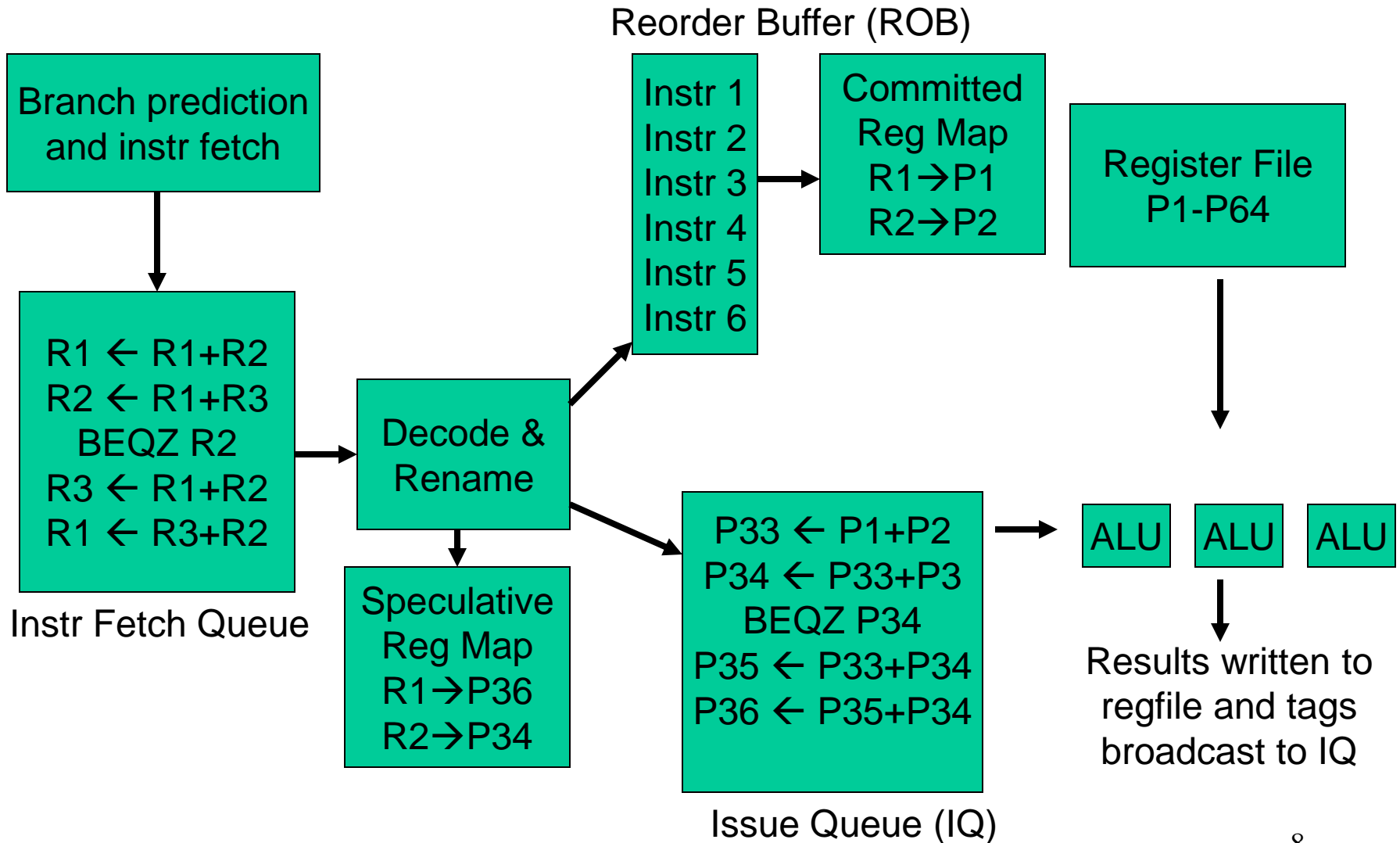
What happens on commit?

# 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

# The Alpha 21264 Out-of-Order Implementation





# 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

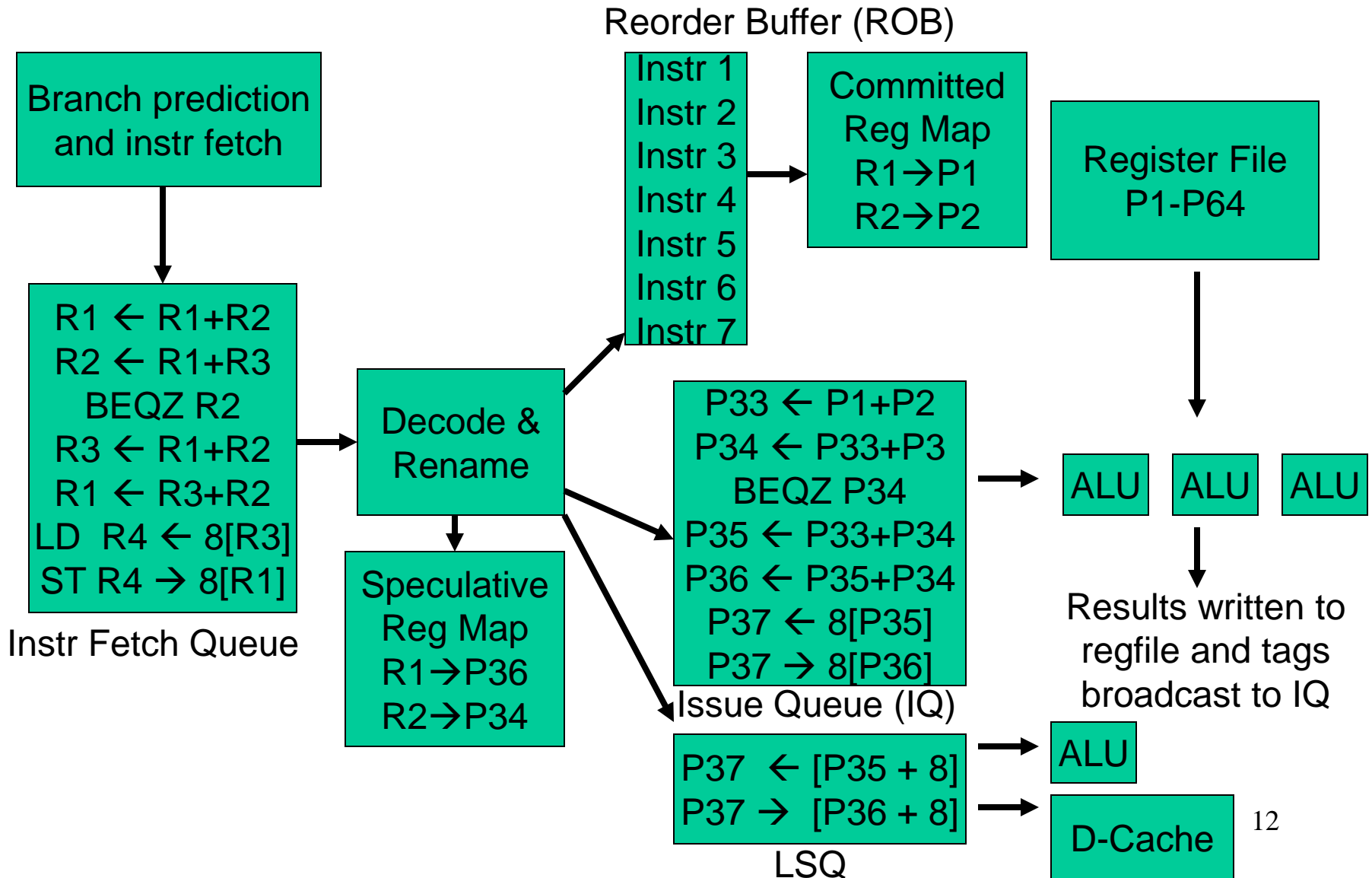
# Memory Dependence Checking

---

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



# Title

---

- Bullet