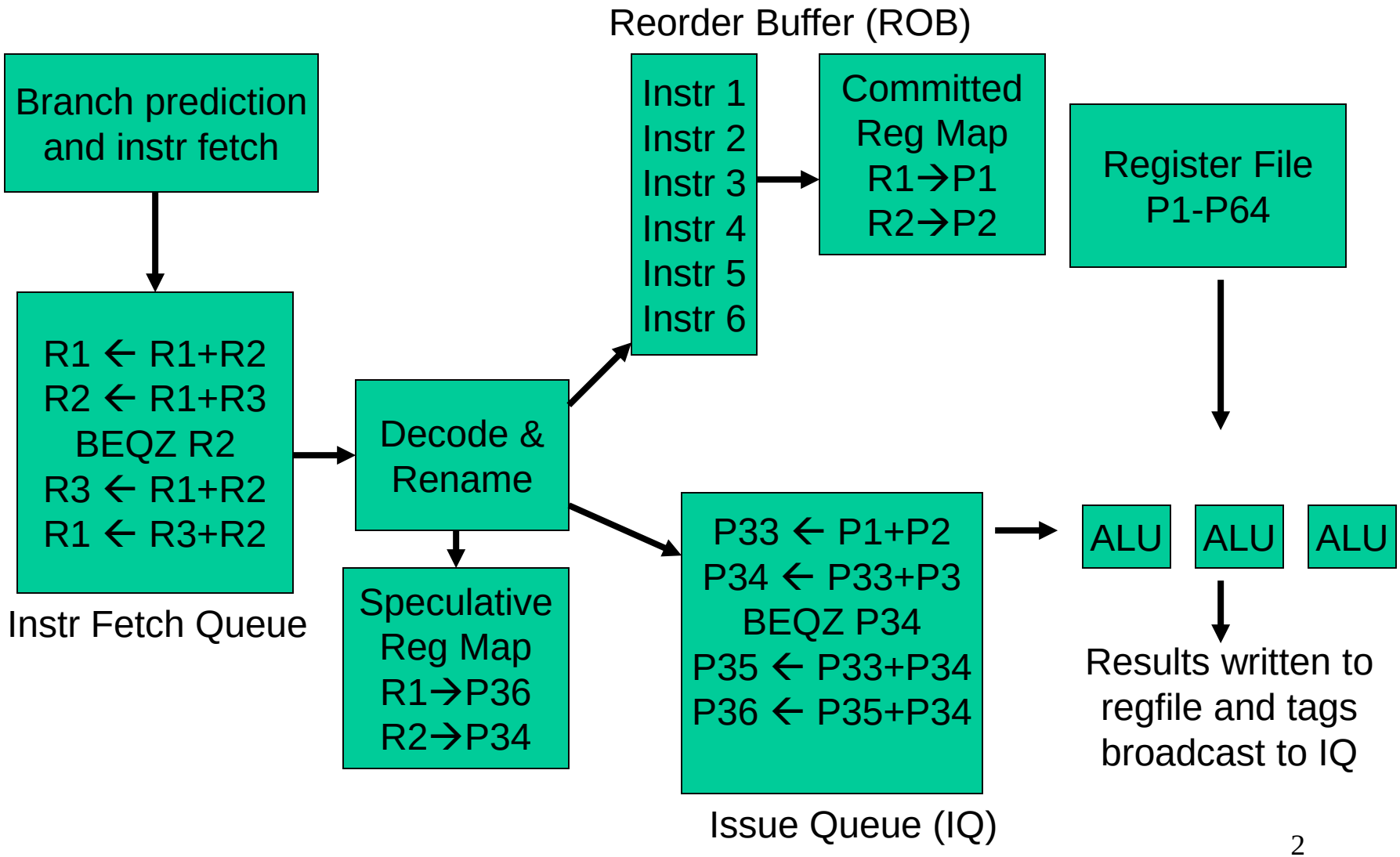
Lecture 16: Core Design

- Today: basics of implementing a correct ooo core:
  register renaming, commit, LSQ, issue queue

The Alpha 21264 Out-of-Order Implementation



Rename

A lr1 ← lr2 + lr3
B lr2 ← lr4 + lr5
C lr6 ← lr1 + lr3
D lr6 ← lr1 + lr2

RAR lr3
RAW lr1
WAR lr2
WAW lr6
A ; BC ; D

pr7 ← pr2 + pr3
pr8 ← pr4 + pr5
pr9 ← pr7 + pr3
pr10 ← pr7 + pr8

RAR pr3
RAW pr7
WAR x
WAW x
AB ; CD

Commit Example

Assume a processor with 6 logical regs and 10 physical regs

			Map Old / New		
A	lr1 ← lr2 + lr3	pr7 ← pr2 + pr3	lr1	pr1	pr7
B	lr2 ← lr4 + lr5	pr8 ← pr4 + pr5	lr2	pr2	pr8
C	lr6 ← lr1 + lr3	pr9 ← pr7 + pr3	lr6	pr6	pr9
D	lr6 ← lr1 + lr2	pr10 ← pr7 + pr8	lr6	pr9	pr10
E	lr3 ← lr6 + lr2	pr1 ← pr10 + pr8	lr3	pr3	pr1
F	lr4 ← lr3 + lr4	pr2 ← pr1 + pr4	lr4	pr4	pr2

Out-of-Order Loads/Stores

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

Memory Dependence Checking

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

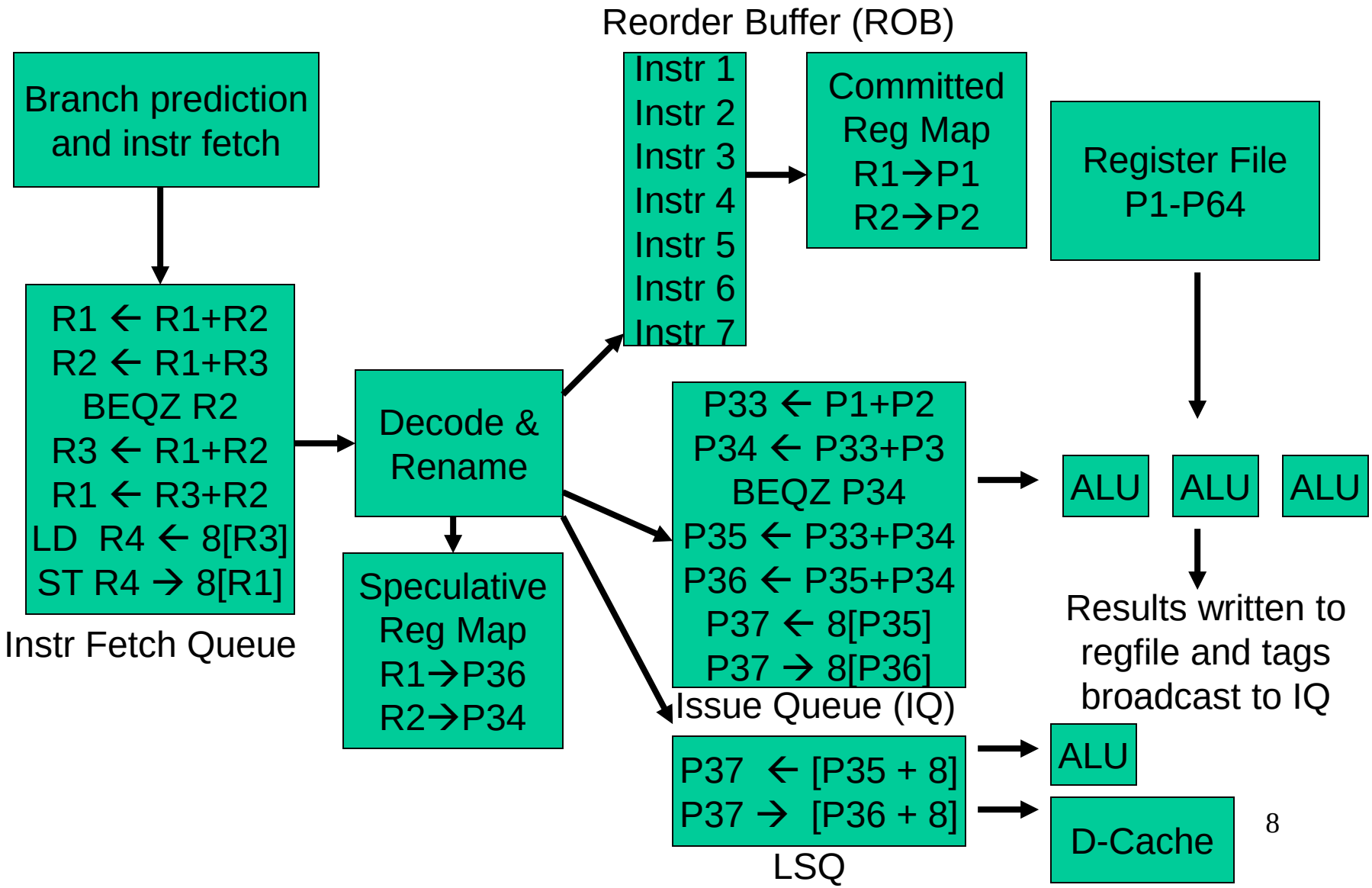
- The issue queue checks for register dependences and executes instructions as soon as registers are ready
- Loads/stores access memory as well – must check for RAW, WAW, and WAR hazards for memory as well
- Hence, first check for register dependences to compute effective addresses; then check for memory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



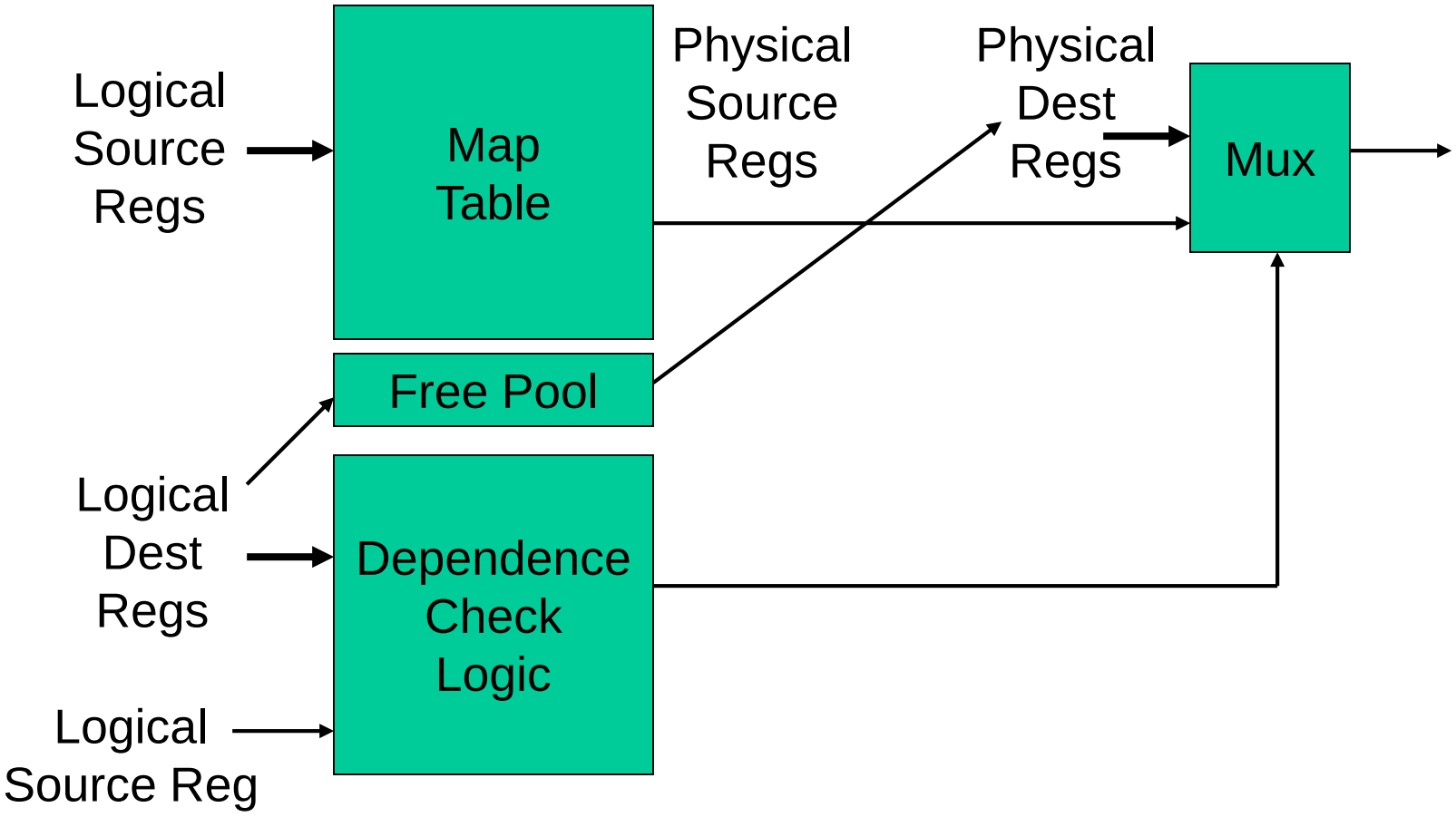
Speculative Issue

- Instr I1 leaves the issue queue at start of cycle 6; the instr then reads operands from the regfile, wires are traversed, instruction executes, result is available at end of cycle 8
- If operand availability is broadcast to issue queue in cycle 9, dependent instruction leaves in cycle 10
- This causes a 4-cycle gap between successive instrs
- Hence, if we know that the instruction takes a cycle to execute, the operand is broadcast to the issue queue in cycle 6 and the dependent instr leaves issue queue in cycle 7; the input operand is correctly bypassed at the FU

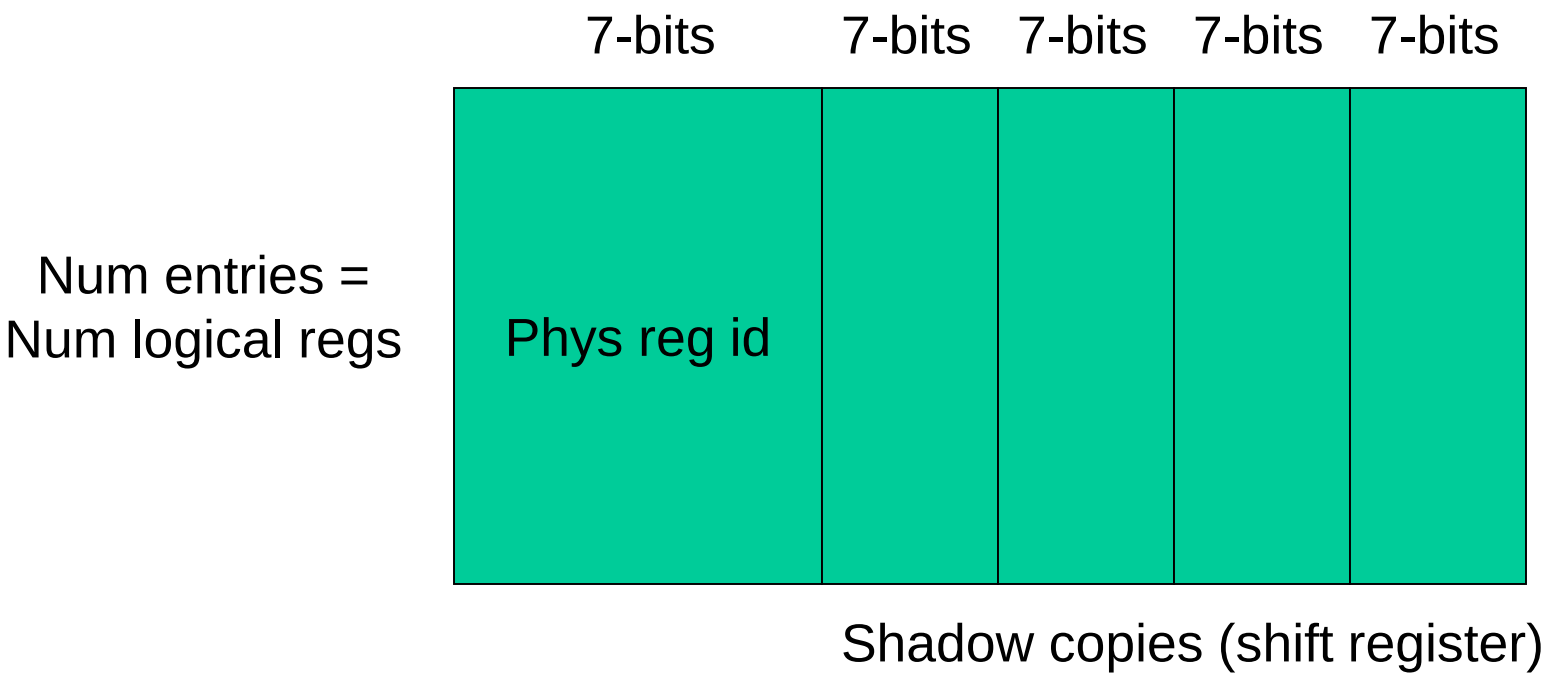
Load Hit Speculation

- The previous optimization assumes that we know the exact latency for every operation
- This is true for all ops except loads (cache hit or miss?)
- Assume hit and schedule accordingly; on a cache miss, must squash all speculatively issued instructions; an instruction therefore sits in the queue until load hits are determined

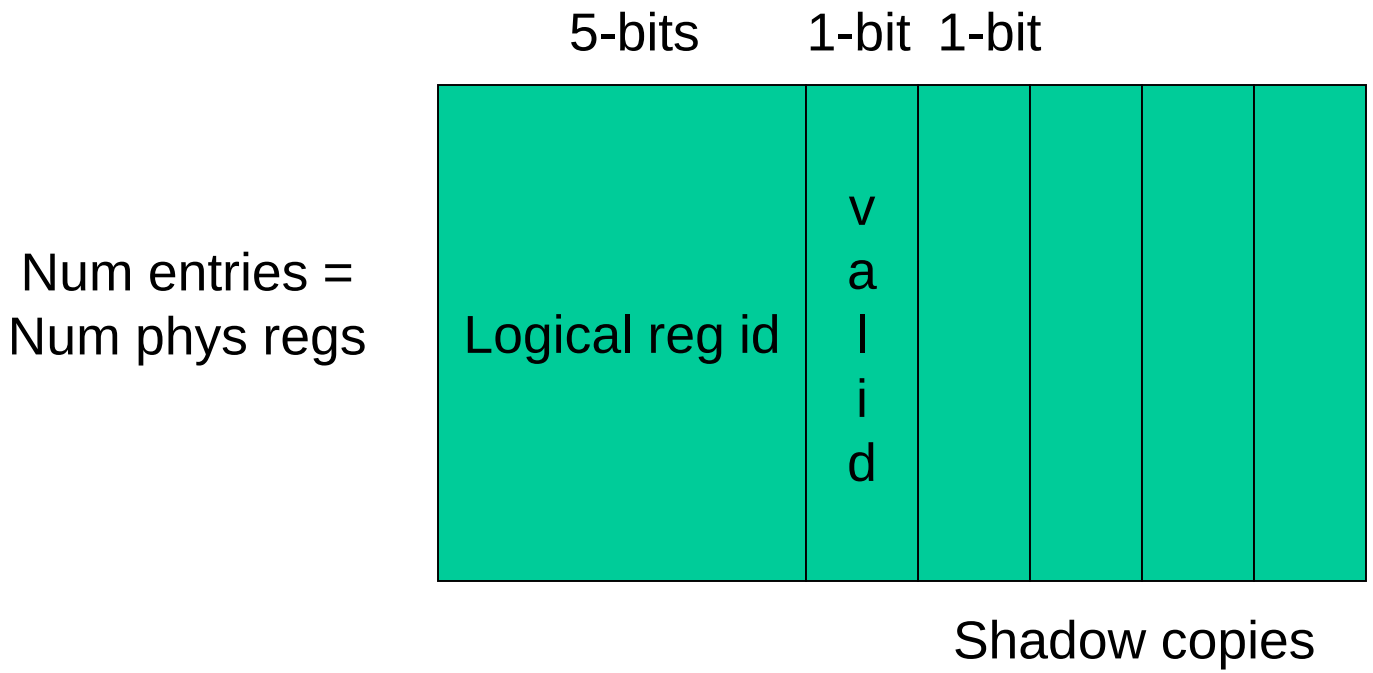
Register Rename Logic



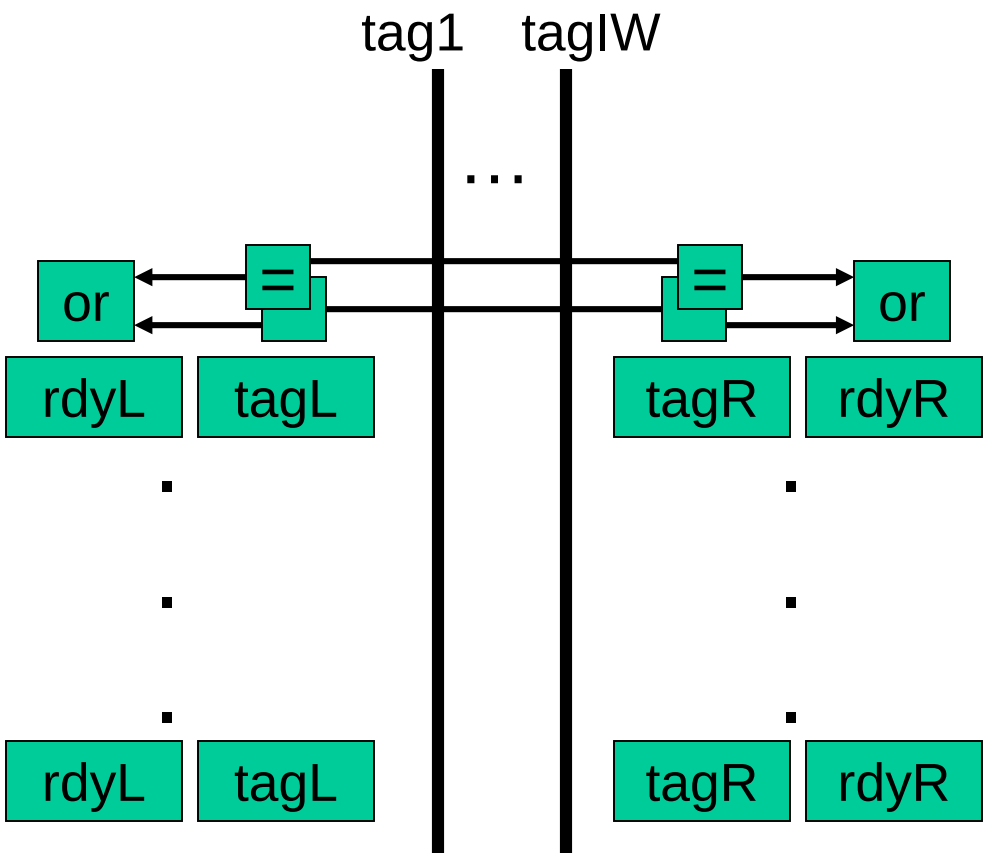
Map Table – RAM



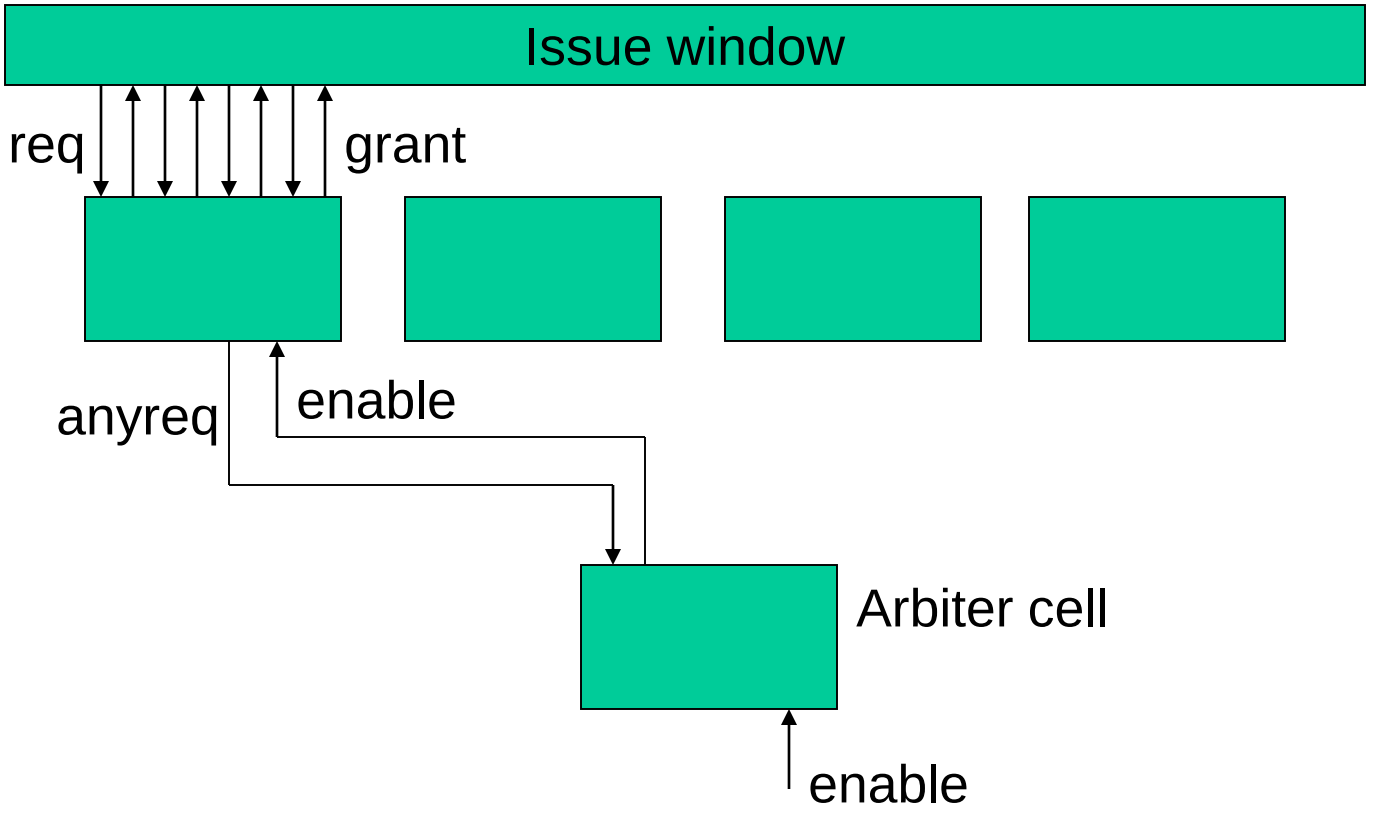
Map Table – CAM



Wakeup Logic



Selection Logic



- For multiple FUs, will need sequential selectors

Structure Complexities

- Critical structures:
  register map tables, issue queue, LSQ, register file, register bypass
- Cycle time is heavily influenced by:
  window size (physical register size), issue width (#FUs)
- Conflict between the desire to increase IPC and clock speed

Title

- Bullet