Lecture: Out-of-order Processors

- Topics: branch predictor wrap-up, a basic out-of-order processor with issue queue, register renaming, and reorder buffer

Amdahl's Law

- Architecture design is very bottleneck-driven – make the common case fast, do not waste resources on a component that has little impact on overall performance/power
- Amdahl's Law: performance improvements through an enhancement is limited by the fraction of time the enhancement comes into play
- Example: a web server spends 40% of time in the CPU and 60% of time doing I/O – a new processor that is ten times faster results in a 36% reduction in execution time (speedup of 1.56) – Amdahl's Law states that maximum execution time reduction is 40% (max speedup of 1.66)

Principle of Locality

- Most programs are predictable in terms of instructions executed and data accessed
- The 90-10 Rule: a program spends 90% of its execution time in only 10% of the code
- Temporal locality: a program will shortly re-visit X
- Spatial locality: a program will shortly visit X+1

Problem 1

- What is the storage requirement for a global predictor that uses 3-bit saturating counters and that produces an index by XOR-ing 12 bits of branch PC with 12 bits of global history?

Problem 1

- What is the storage requirement for a global predictor that uses 3-bit saturating counters and that produces an index by XOR-ing 12 bits of branch PC with 12 bits of global history?

  The index is 12 bits wide, so the table has 2^12 saturating counters. Each counter is 3 bits wide. So total storage = 3 * 4096 = 12 Kb or 1.5 KB

Problem 2

- What is the storage requirement for a tournament predictor that uses the following structures:
  - a “selector” that has 4K entries and 2-bit counters
  - a “global” predictor that XORs 14 bits of branch PC with 14 bits of global history and uses 3-bit counters
  - a “local” predictor that uses an 8-bit index into L1, and produces a 12-bit index into L2 by XOR-ing branch PC and local history. The L2 uses 2-bit counters.

Problem 2

- What is the storage requirement for a tournament predictor that uses the following structures:
  - a “selector” that has 4K entries and 2-bit counters
  - a “global” predictor that XORs 14 bits of branch PC with 14 bits of global history and uses 3-bit counters
  - a “local” predictor that uses an 8-bit index into L1, and produces a 12-bit index into L2 by XOR-ing branch PC and local history. The L2 uses 2-bit counters.

Selector = 4K * 2b = 8 Kb
Global = 3b * 2^14 = 48 Kb
Local = (12b * 2^8) + (2b * 2^12) = 3 Kb + 8 Kb = 11 Kb
Total = 67 Kb

Problem 3

- For the code snippet below, estimate the steady-state bpred accuracies for the default PC+4 prediction, the 1-bit bimodal, 2-bit bimodal, global, and local predictors. Assume that the global/local preds use 5-bit histories.

```
do {
    for (i=0; i<4; i++) {
       increment something
    }
    for (j=0; j<8; j++) {
       increment something
    }
    k++;
} while (k < some large number)
```

Problem 3

- For the code snippet below, estimate the steady-state bpred accuracies for the default PC+4 prediction, the 1-bit bimodal, 2-bit bimodal, global, and local predictors. Assume that the global/local preds use 5-bit histories.

```
do {
    for (i=0; i<4; i++) {
       increment something
    }
    for (j=0; j<8; j++) {
       increment something
    }
    k++;
} while (k < some large number)
```

PC+4: 2/13 = 15%

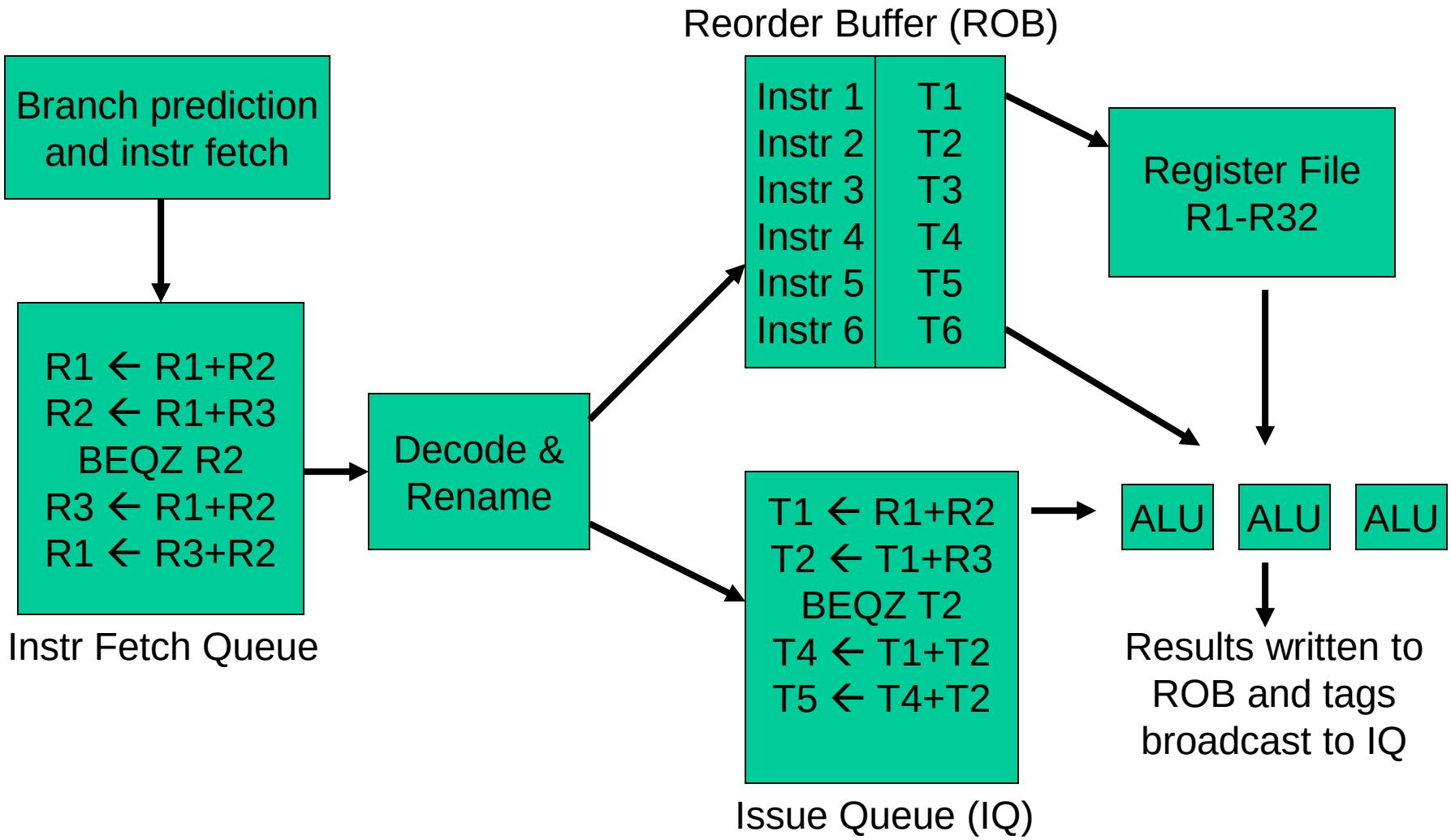
1b Bim: (2+6+1)/(4+8+1)
= 9/13 = 69%

2b Bim: (3+7+1)/13
= 11/13 = 85%

Global: (4+7+1)/13
= 12/13 = 92%
(gets confused by 01111 unless you take branch-PC into account while indexing)

Local: (4+7+1)/13
= 12/13 = 92%

An Out-of-Order Processor Implementation



Problem 1

- Show the renamed version of the following code:
  Assume that you have 4 rename registers T1-T4

R1 ← R2+R3
R3 ← R4+R5
BEQZ R1
R1 ← R1 + R3
R1 ← R1 + R3
R3 ← R1 + R3

Problem 1

- Show the renamed version of the following code:
  Assume that you have 4 rename registers T1-T4

R1 ← R2+R3	T1 ← R2+R3
R3 ← R4+R5	T2 ← R4+R5
BEQZ R1	BEQZ T1
R1 ← R1 + R3	T4 ← T1+T2
R1 ← R1 + R3	T1 ← T4+T2
R3 ← R1 + R3	T2 ← T1 +R3

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

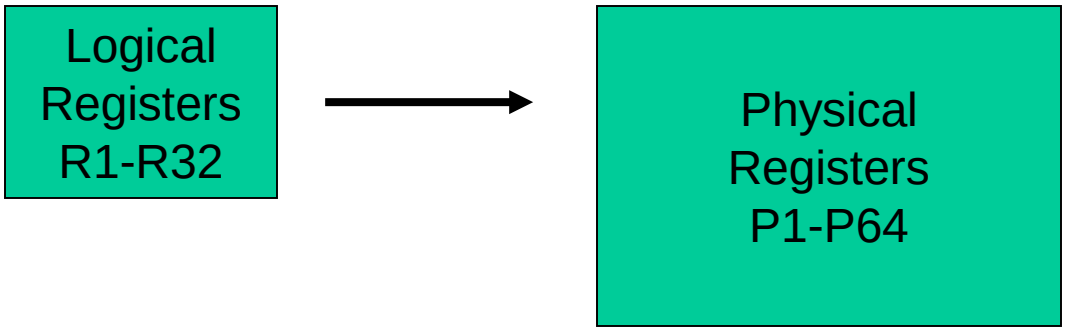
- If instr-3 raises an exception, wait until it reaches the top of the ROB – at this point, R1-R32 contain results for all instructions up to instr-3 – save registers, save PC of instr-3, and service the exception

- 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)

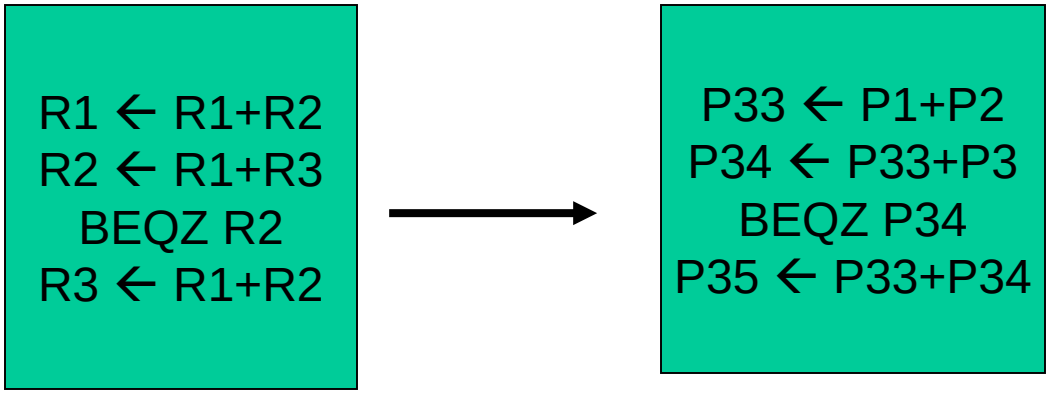
- Potential problems: ?

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

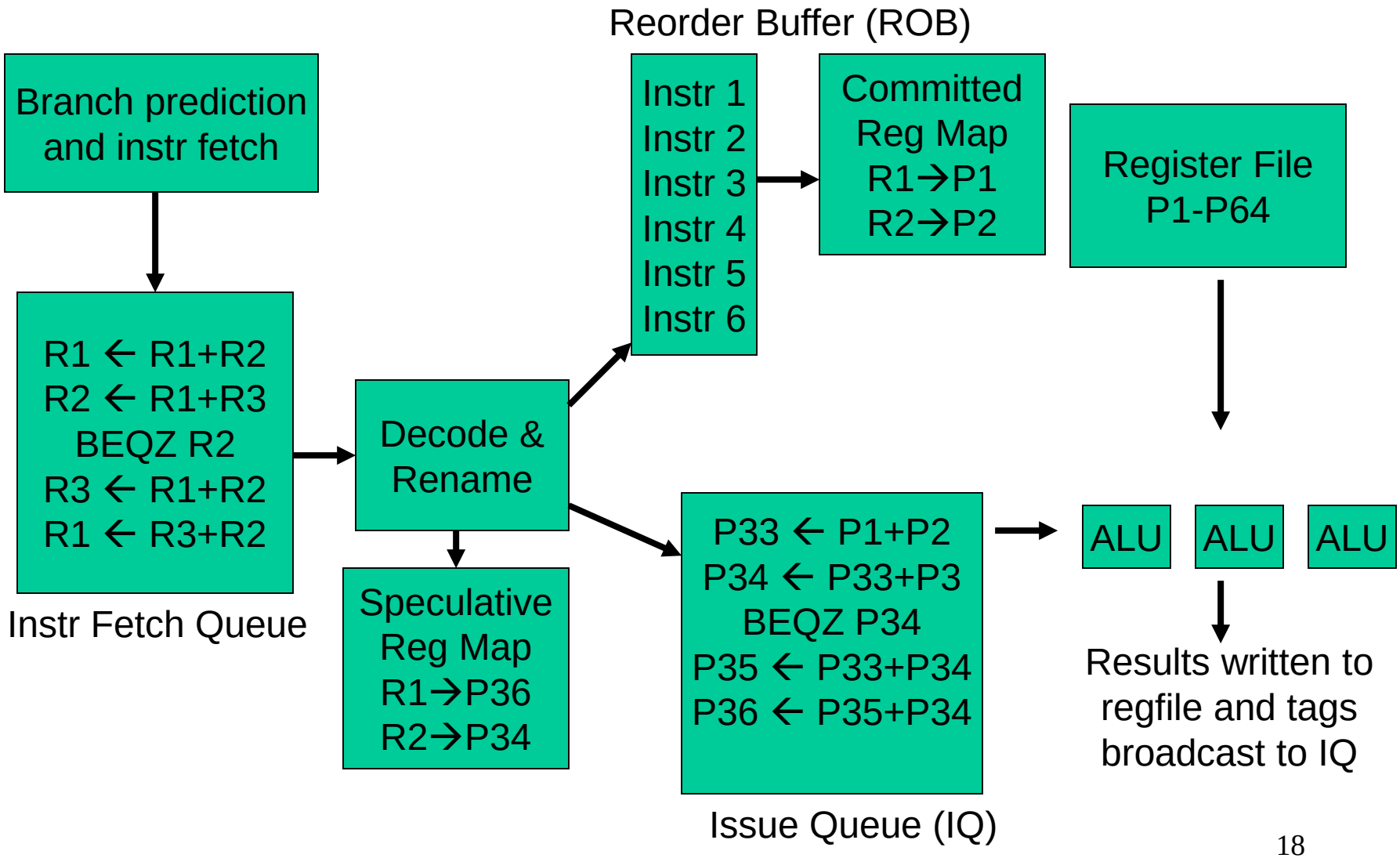


What happens on commit?

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



Problem 2

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

R1 ← R2+R3
R3 ← R4+R5
BEQZ R1
R1 ← R1 + R3
R1 ← R1 + R3
R3 ← R1 + R3
R4 ← R3 + R1

Problem 2

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

```
R1 ← R2+R3                P33 ← P2+P3
R3 ← R4+R5                P34 ← P4+P5
BEQZ  R1                  BEQZ P33
R1 ← R1 + R3              P35 ← P33+P34
R1 ← R1 + R3              P36 ← P35+P34
R3 ← R1 + R3              P1  ← P36+P34
R4 ← R3 + R1              P3  ← P1+P36
```

Title

- Bullet