EECS 470

Adding a RoB

Lecture 7 – Winter 2024

Slides developed in part by Profs. Austin, Brehob, Falsafi, Hill, Hoe, Lipasti, Martin, Roth, Shen, Smith, Sohi, Tyson, Vijaykumar, and Wenisch of Carnegie Mellon University, Purdue University, University of Michigan, University of Pennsylvania, and University of Wisconsin.

Last time:

- Covered branch predictors
  - Direction
    - Bimodal, local history, global, gshare, tournament
  - Address
    - BTB
    - RAS (briefly)

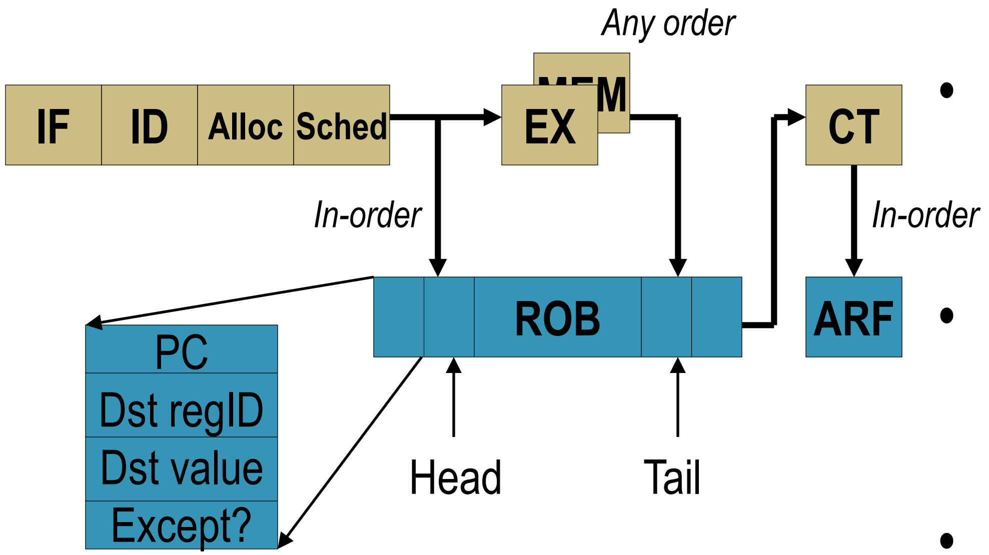
General speculation

- Control speculation
  - “I think this branch will go to address 90004”
- Data speculation
  - “I’ll guess the result of the load will be zero”
- Memory conflict speculation
  - “I don’t think this load conflicts with any proceeding store.”
- Error speculation
  - “I don’t think there were any errors in this calculation”

Speculation in general

- Need to be 100% sure on final correctness!
  - So need a recovery mechanism
  - Must make forward progress!
- Want to speed up overall performance
  - So recovery cost should be low or ***expected*** rate of occurrence should be low.
  - There can be a real trade-off on *accuracy*, *cost of recovery*, and *speedup when correct.*
- Should keep the worst case in mind…

Precise Interrupts and branches via the Reorder Buffer



- Reorder Buffer (ROB)
  - Circular queue of spec state
  - May contain multiple definitions of *same* register

- @ **Alloc**
  - Allocate result storage at Tail
- @ **Sched**
  - Get inputs (ROB T-to-H then ARF)
  - Wait until all inputs ready
- @ **WB**
  - Write results/fault to ROB
  - Indicate result is ready
- @ **CT**
  - Wait until inst @ Head is done
  - If fault, initiate handler
  - Else, write results to ARF
  - Deallocate entry from ROB

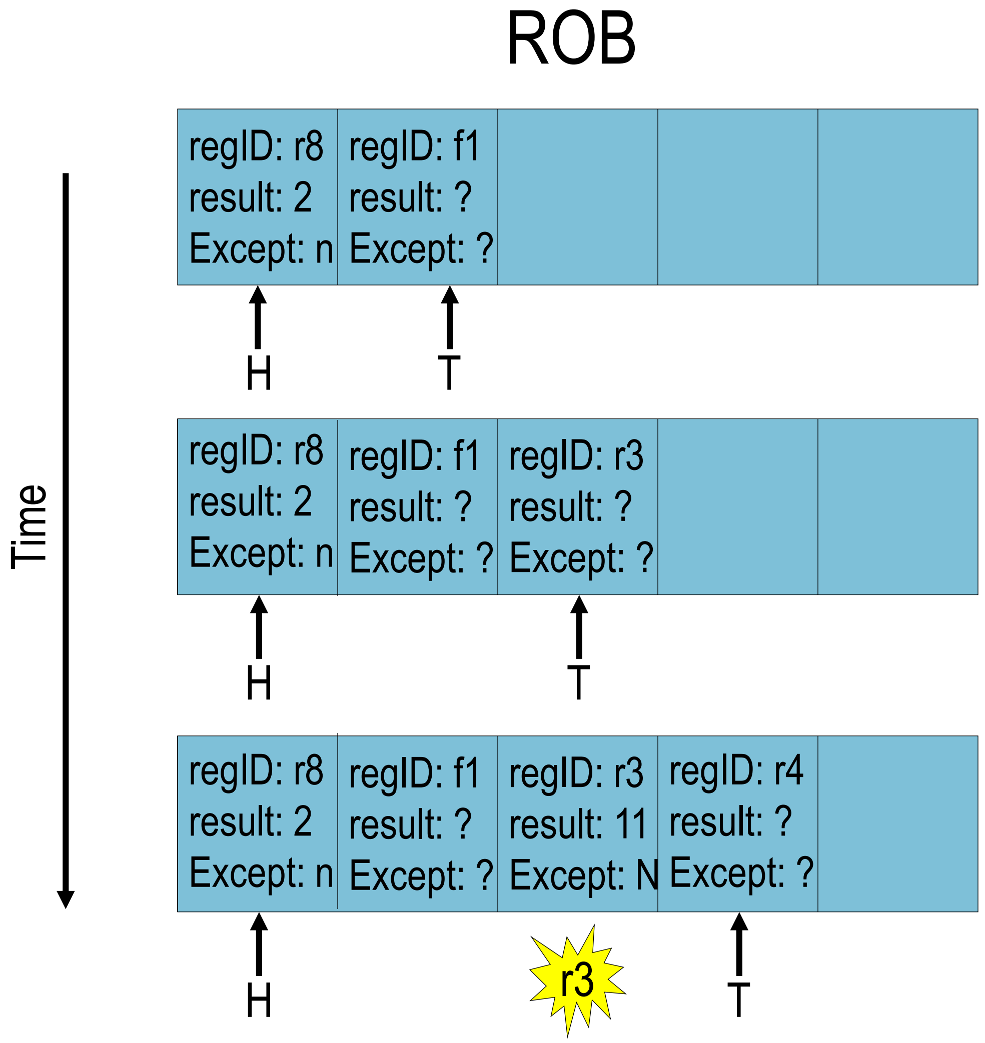
Reorder Buffer Example

Code Sequence

f1 = f2 / f3
r3 = r2 + r3
r4 = r3 – r2

Initial Conditions

- reorder buffer empty
- f2 = 3.0
- f3 = 2.0
- r2 = 6
- r3 = 5



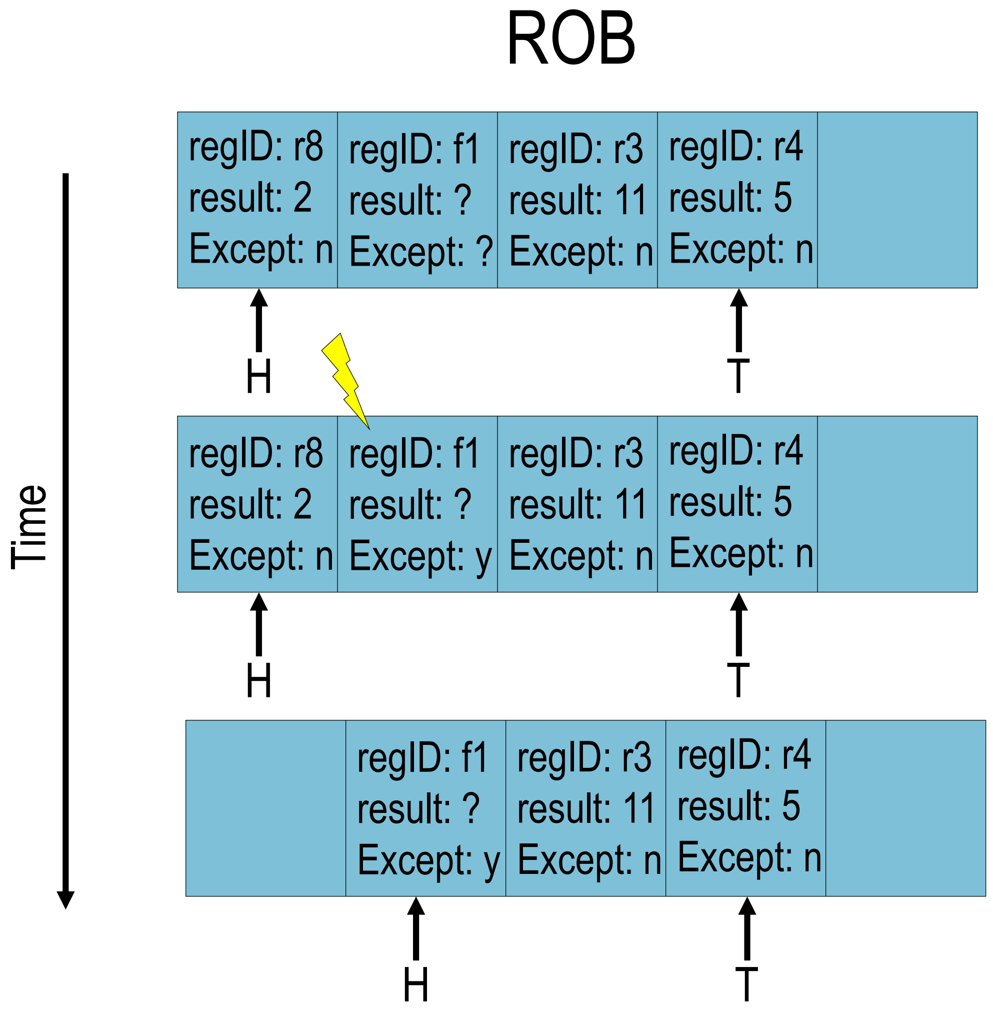
Reorder Buffer Example

<u>Code Sequence</u>

f1 = f2 / f3
r3 = r2 + r3
r4 = r3 – r2

Initial Conditions

- reorder buffer empty
- f2 = 3.0
- f3 = 2.0
- r2 = 6
- r3 = 5



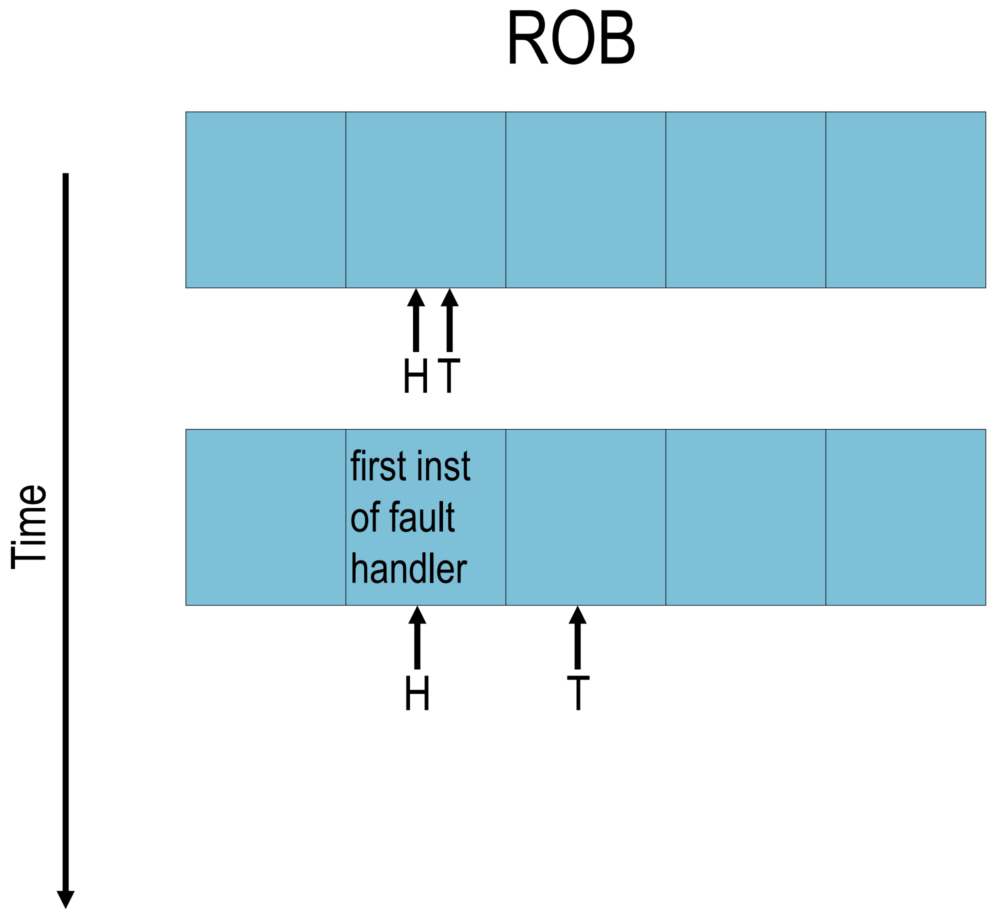
Reorder Buffer Example

Code Sequence

f1 = f2 / f3
r3 = r2 + r3
r4 = r3 – r2

Initial Conditions

- reorder buffer empty
- f2 = 3.0
- f3 = 2.0
- r2 = 6
- r3 = 5



There is more complexity here

- Rename table needs to be cleared
  - Everything is in the ARF
  - Really do need to finish everything which was before the faulting instruction in program order.
- What about branches?
  - Would need to drain everything before the branch.
    - Why not just squash everything that follows it?

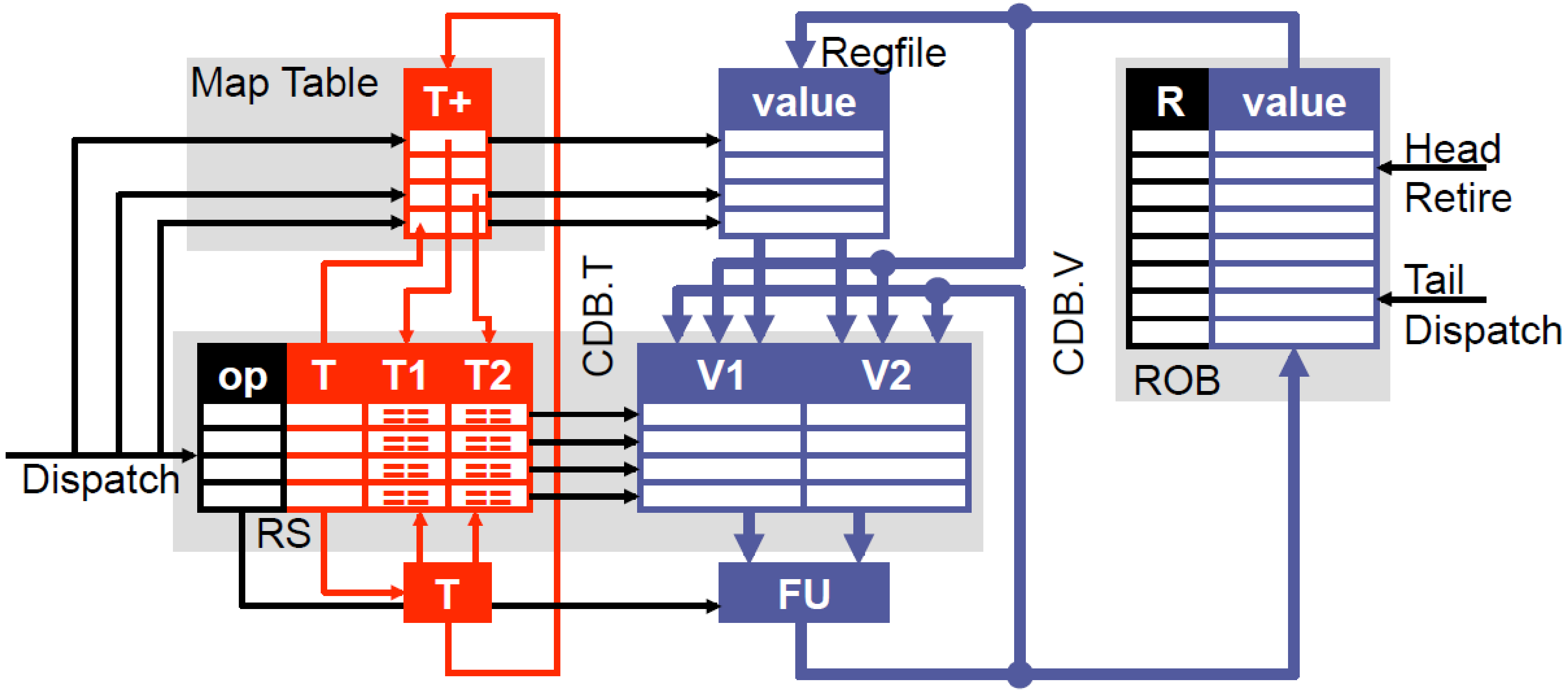
And while we’re at it…

- Does the ROB replace the RS?
  - Is this a good thing? Bad thing?

ROB

- ROB
  - ROB is an *in-order* queue where instructions are placed.
  - Instructions *complete* (retire) in-order
  - Instructions still *execute* out-of-order
  - Still use RS
    - Instructions are issued to RS and ROB at the same time
    - Rename is to ROB entry, not RS.
    - When *execute* done instruction leaves RS
  - Only when all instructions in before it in program order are done does the instruction retire.

Adding a Reorder Buffer



Tomasulo Data Structures

(Timing Free Example, “P6 scheme”)

CDB	
T	V

Map Table	
Reg	Tag
r0	
r1	
r2	
r3	
r4	

Reservation Stations (RS)								
T	FU	busy	op	RoB	T1	T2	V1	V2
1								
2								
3								
4								
5								

ARF	
Reg	V
r0	
r1	
r2	
r3	
r4	

Instruction
r0=r1*r2
r1=r2*r3
Branch if r1=0
r0=r1+r1
r2=r2+1

Reorder Buffer (RoB)							
RoB Number	0	1	2	3	4	5	6
Dest. Reg.							
Value							

Review Questions

- Could we make this work without the RS?
  - If so, why do we do that?
- Why is it important to retire in order?
- Why must branches wait until retirement before they announce their mispredict?
  - Any other ways to do this?

More review questions

1. What is the purpose of the RoB?
2. Why do we have both a RoB and a RS?
   - Yes, that was pretty much on the last page…
3. Misprediction
   a) When to we resolve a mis-prediction?
   b) What happens to the main structures (RS, RoB, ARF, Rename Table) when we mispredict?
4. What is the whole purpose of OoO execution?

And yet more review questions!

1. What is the purpose of the RoB?
2. Why do we have both a RoB and a RS?
3. Misprediction
    a) When to we resolve a mis-prediction?
    b) What happens to the main structures (RS, RoB, ARF, Rename Table) when we mispredict?
4. What is the whole purpose of OoO execution?

When an instruction is *dispatched* how does it impact each major structure?

- Rename table?
- ARF?
- RoB?
- RS?

When an instruction *completes execution* how does it impact each major structure?

- Rename table?
- ARF?
- RoB?
- RS?

When an instruction *retires* how does it impact each major structure?

- Rename table?
- ARF?
- RoB?
- RS?

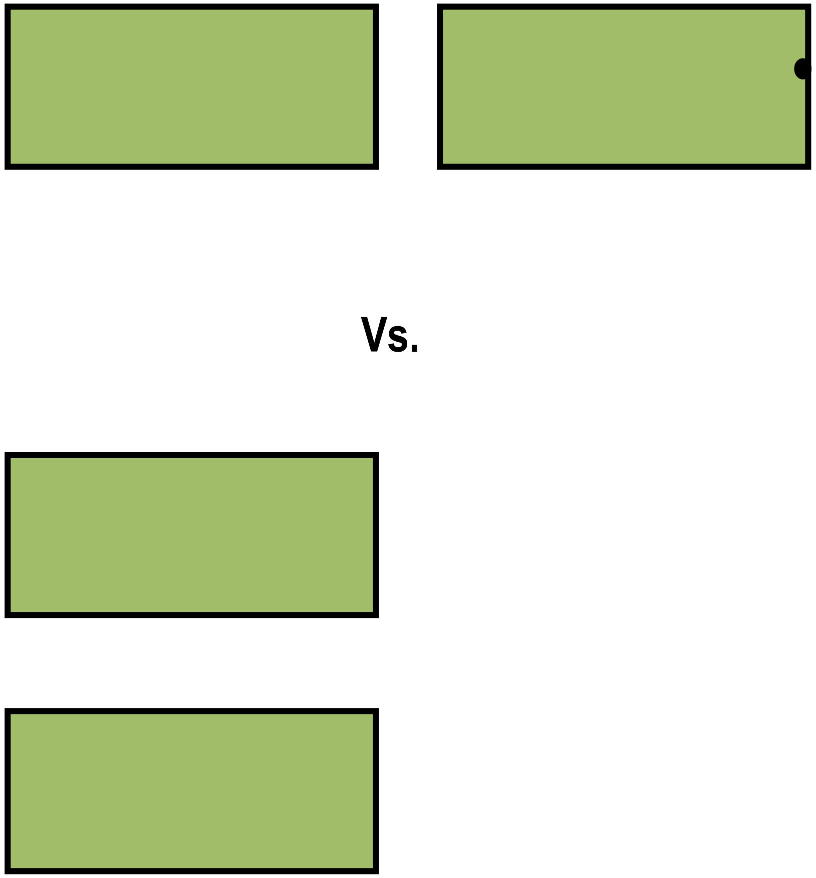
Topic change

- Why on earth are we doing this?
  - Why do we think it helps?
- Homework 2 problems 5 and 6 made the argument.
  - Only need to obey true data dependencies.
    - Huge speedup ***potential***.

Optimizing CPU Performance

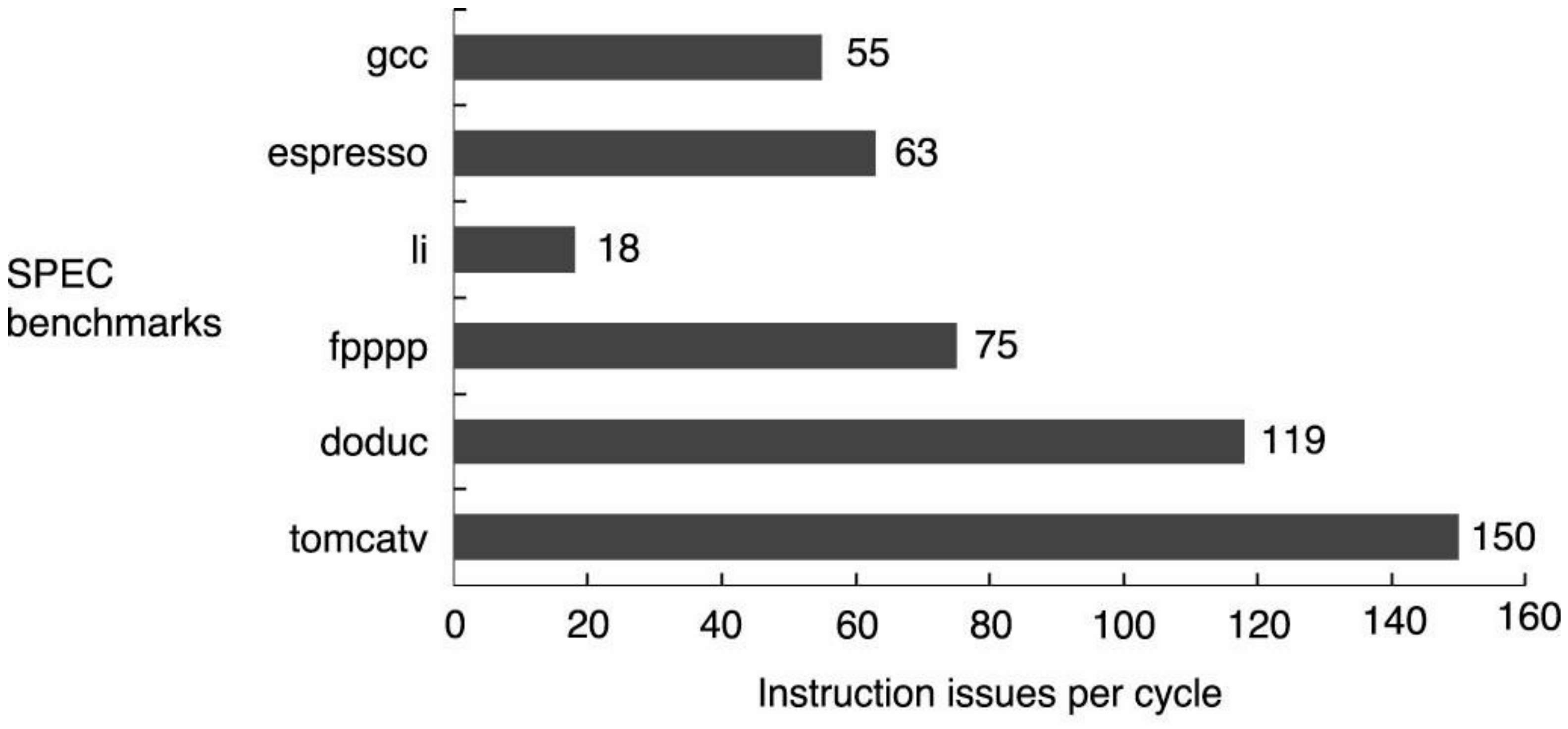
- Golden Rule: $t_{CPU} = N_{inst} * CPI * t_{CLK}$
- Given this, what are our options
  - Reduce the number of instructions executed
  - Reduce the cycles to execute an instruction
  - Reduce the clock period
- Our first focus: Reducing CPI
  - Approach: *Instruction Level Parallelism* (ILP)

Why ILP?

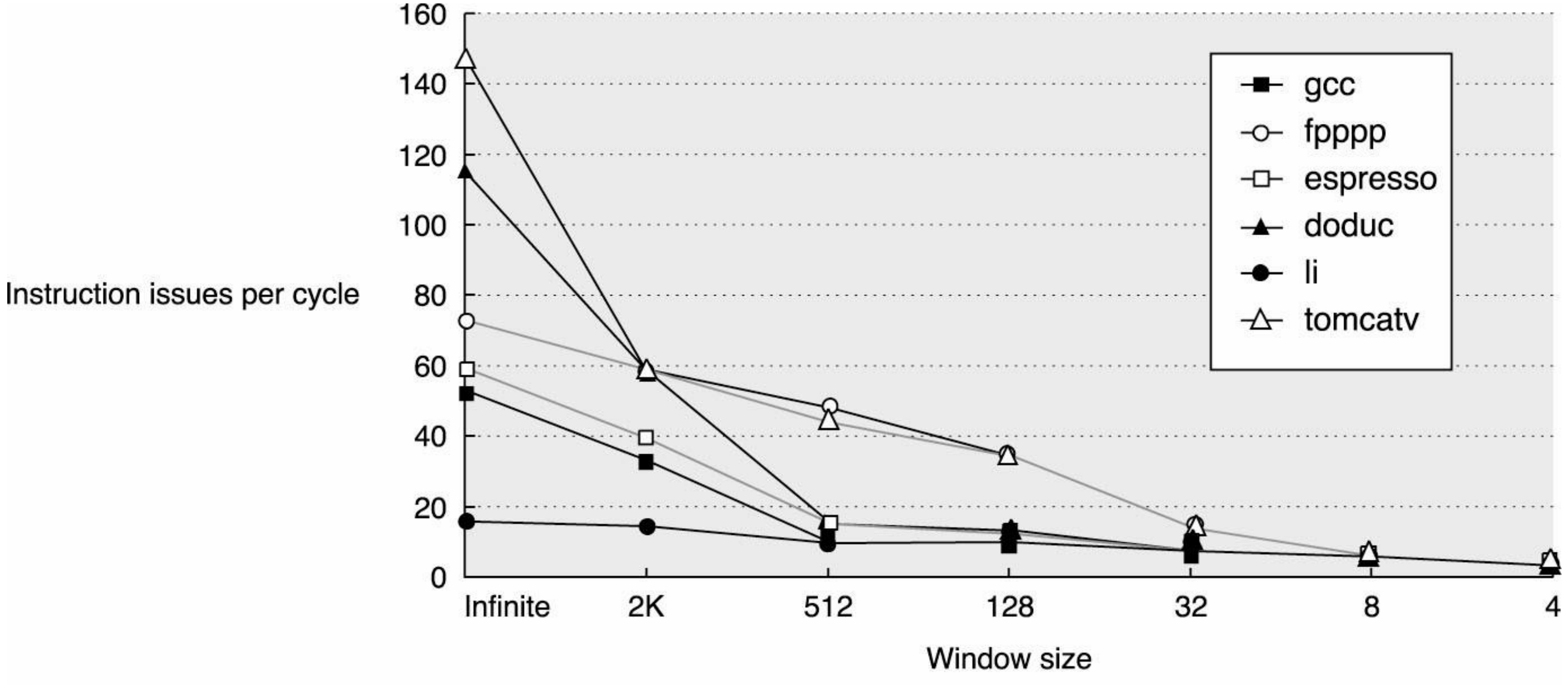


- Requirements
  - Parallelism
  - Large window
  - Limited control deps
  - Eliminate “false” deps
  - Find run-time deps

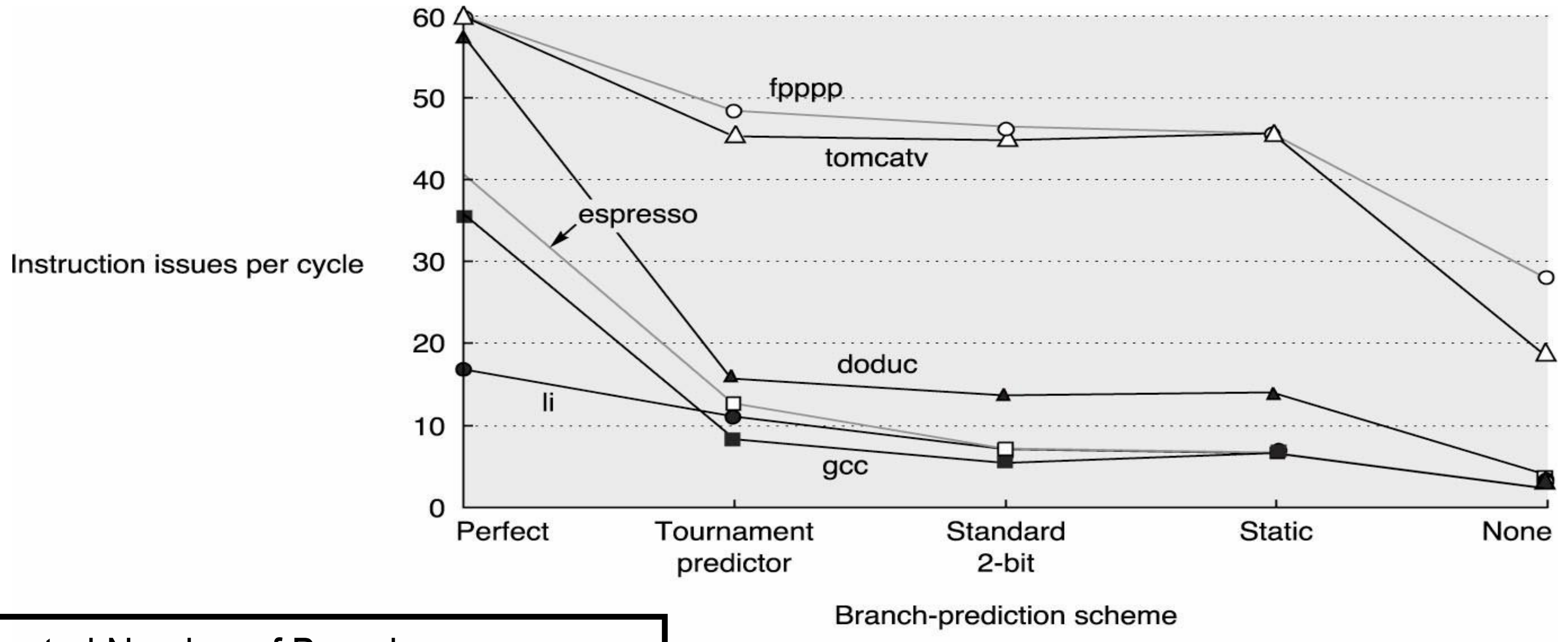
How Much ILP is There? (Chapter 3.10)



How Large Must the “Window” Be?



ALU Operation *GOOD*, Branch *BAD*

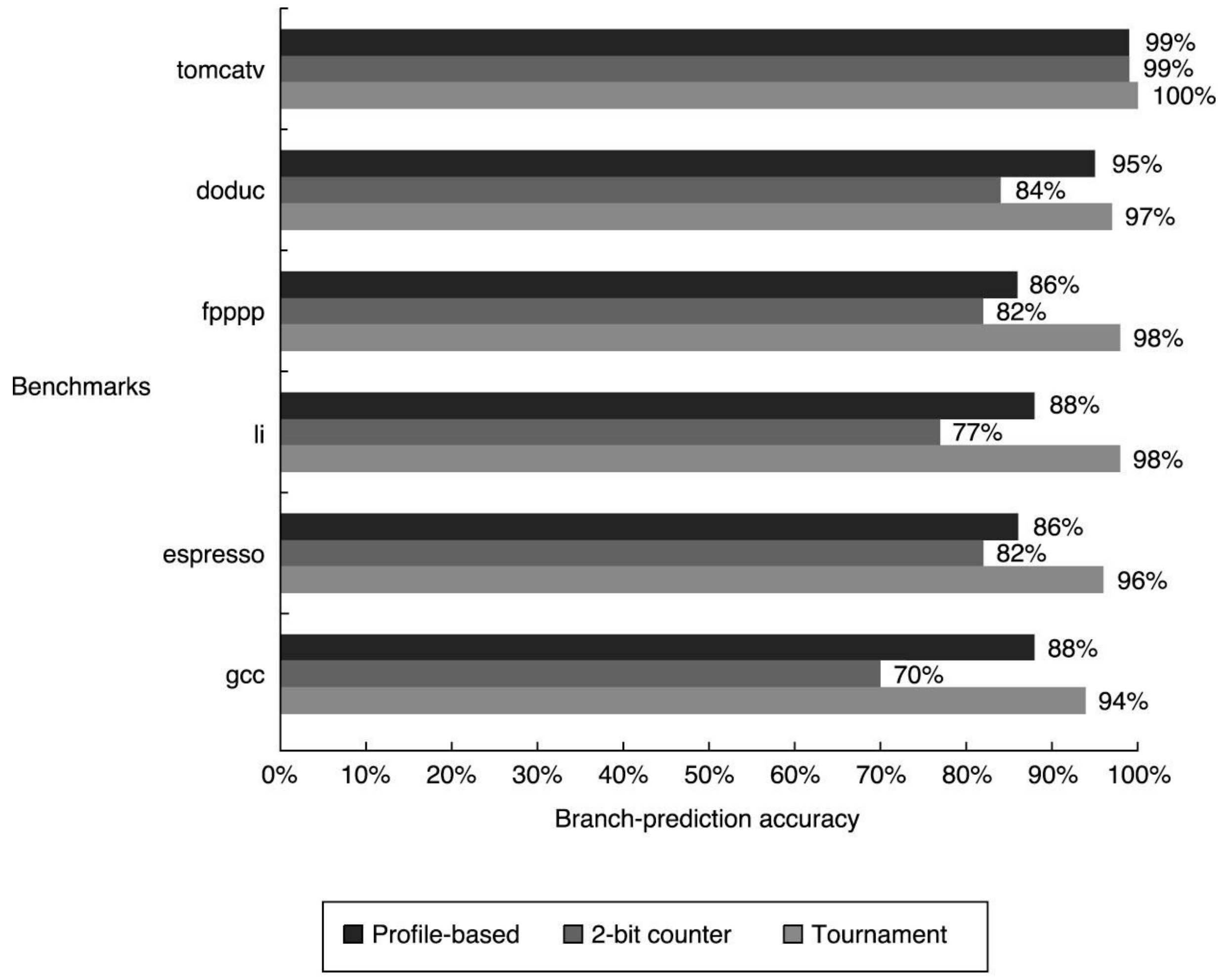


<u>Expected Number of Branches Between Mispredicts</u>

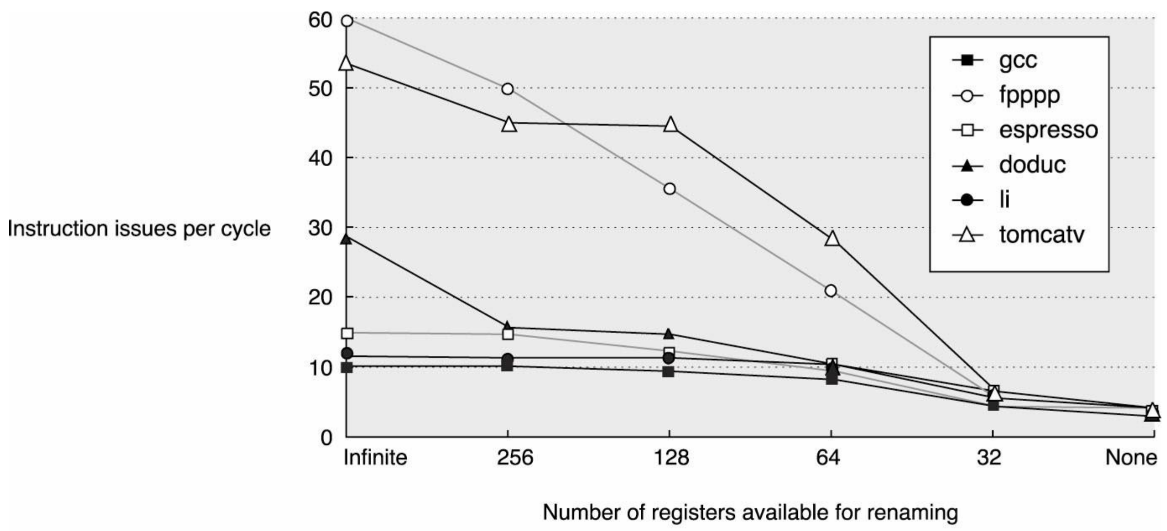
E(X) ~ 1/(1-p)

E.g., p = 95%, E(X) ~ 20 brs, 100-ish insts

How Accurate are Branch Predictors?



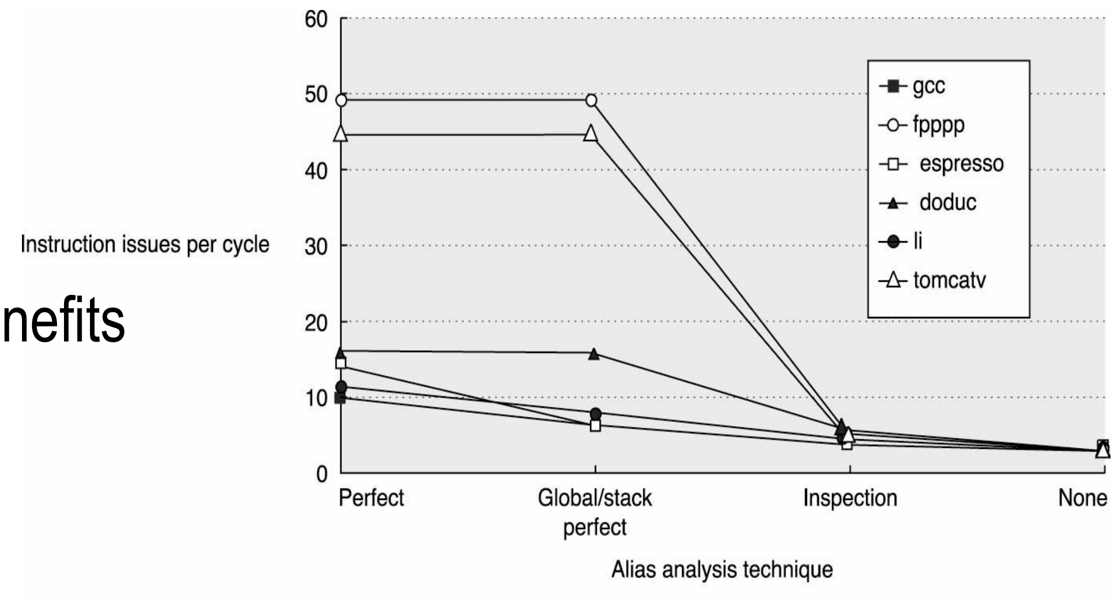
Impact of Physical Storage Limitations



- Each instruction “in flight” must have storage for its result
  - Really worse than this because of mispeculation…

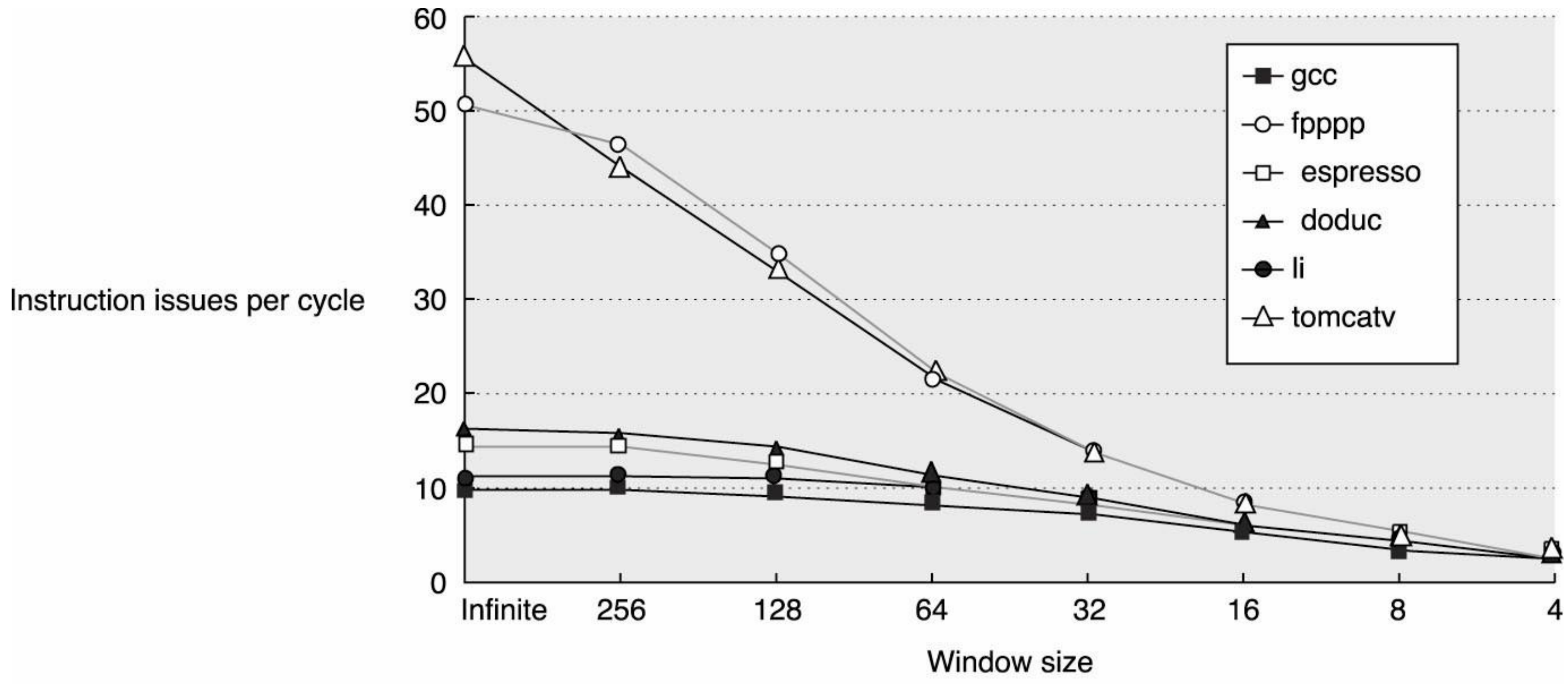
Registers *GOOD*, Memory *BAD*

- Benefits of registers
  - Well described deps
  - Fast access
  - Finite resource
- Memory loses these benefits for flexibility



$$\begin{tikzcd}
{*p = \ldots} \arrow[dd, "?"] \\
{*q = \ldots} \arrow[d] \\
{\ldots = *p}
\end{tikzcd}$$

“Bottom Line” for an Ambitious Design


Instruction issues per cycle
Infinite
Window size
gcc
fpppp
espresso
doduc
li
tomcatv