EECS 470

Final touches on Out-of-Order execution

Review
Tclk
Superscalar
Looking back
Looking forward

Lecture 10 – 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.

Times and Dates and Places

- Exam Q&A
  - *Today 6-8 pm 1200 EECS*
- In class on Thursday similar Q&A.
- One from yesterday should be in the lecture recordings by the end of the day.

- Midterm Thursday 7-9pm
  - DOW 1010 and 1014
  - Assignments posted shortly

- If you have a conflict or other issue, be sure we are aware of it.

RAT

AR	Target
0	4
1	2
2	7
3	1

A: R1=MEM[R2+0]
B: R2=R1/R3
C: R3=R2+R0
D: Branch (R1==0)
E: R3=R1+R3
F: R3=R3+R0
G: R3=R3+19
H: R1=R7+R6

RRAT

AR	Target
0	
1	
2	
3	

RoB

PRF

0	1	2	3	4	5	6	7	8	9
3	2	44	55	3	66	7	11	8	20

Alternative option (v0.9?)

- Use “back pointers” instead of RRAT.
  - Record, in the ROB, which value in the RAT you overwrote.
    - On commit, free that value (it will be the same as the one you would have overwritten in the RAT!)
    - On mispredict, “undo” each step in reverse order (from tail to head).
  - This gives same functionality as RRAT.
    - Slower to handle mispredict ***that is at the head of the RoB.***

R10K discussion

- What do the RAT entries point to?
- What do the RRAT entries point to?
- When do we write a value to the PRF?
- When do we write a value to the RRAT?
- What happens on a mispredict?

RAT

AR	Target
0	4
1	2
2	7
3	1

A: R1=MEM[R2+0]
B: R2=R1/R3
C: R3=R2+R0
D: Branch (R1==0)
E: R3=R1+R3
F: R3=R3+R0
G: R3=R3+19
H: R1=R7+R6

RRAT

AR	Target
0	
1	
2	
3	

RoB

PRF

0	1	2	3	4	5	6	7	8	9
3	2	44	55	3	66	7	11	8	20

What happens on… (R10K)

- Dispatch
- Execute Complete?
- Commit?
- Commit of a mispredicted branch?

Terminology and background

- Branch RAT (BRAT)
  - “Branch Stack” used by MIPSR10K paper
- Retirement RAT (RRAT)
  - Retirement Map table
  - Architected Map table

Other little details

- We've largely ignored timing in this class.
  - Focus on algorithm, not implementation.
- However, there are some timing issues to worry about.
  - (btw, there are timing slides on the website. Don't take them as "truth" for your project, merely one implementation).
  - One issue is "ships passing in the night"
    - When a dispatching instruction is dependent on data on the CDB in the same cycle.
    - Add a MUX… (where?)

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)

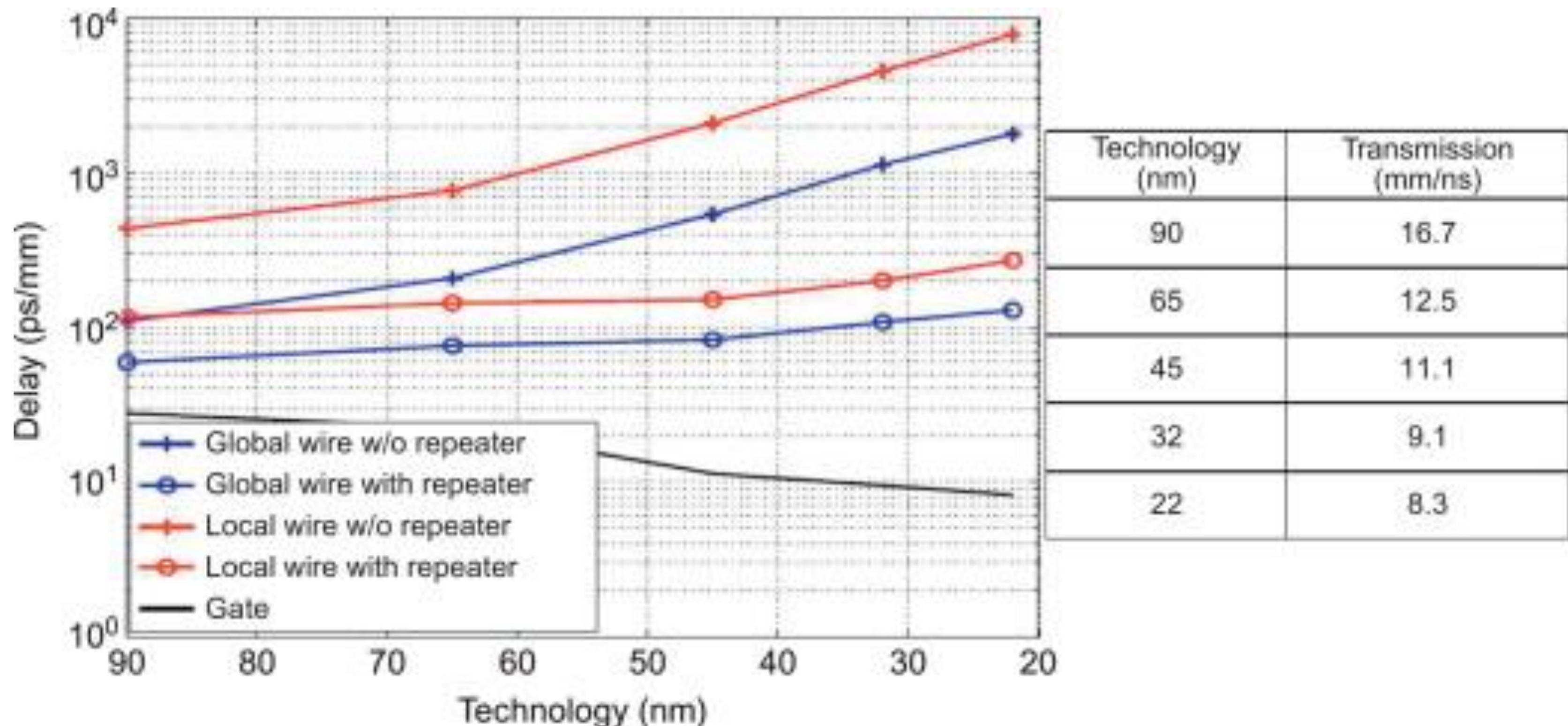
Tclock

(again)

t_{CLK} (again)

- Recall: $t_{CPU} = N_{inst}*CPI*t_{CLK}$
- What defines t_{CLK}?
  - Critical path latency (= logic + wire latency)
  - Latch latency
  - Clock skew
  - Clock period design margins
- In current and future generation designs
  - Wire latency becoming dominant latency of critical path
  - Due to growing side-wall capacitance
  - Brings a spatial dimension to architecture optimization
    - E.g., How long are the wires that will connect these two devices?

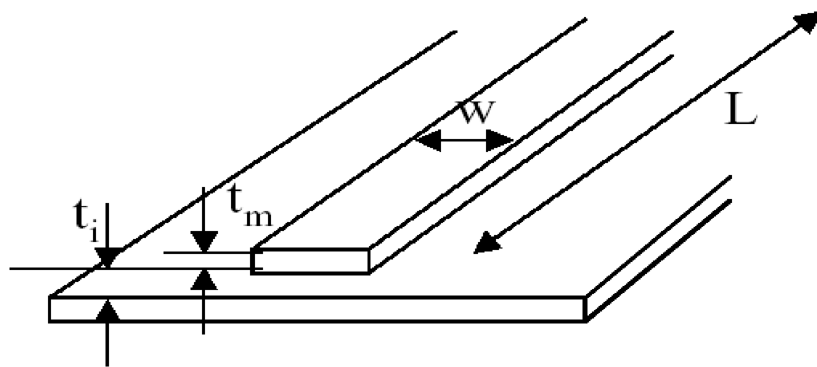
Wire delay vs gate delay



Technology (nm)	Transmission (mm/ns)
90	16.7
65	12.5
45	11.1
32	9.1
22	8.3

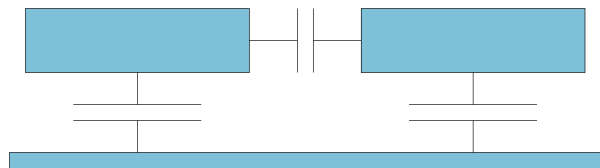
From https://www.sciencedirect.com/science/article/pii/B9780128009796000044

Determining the Latency of a Wire

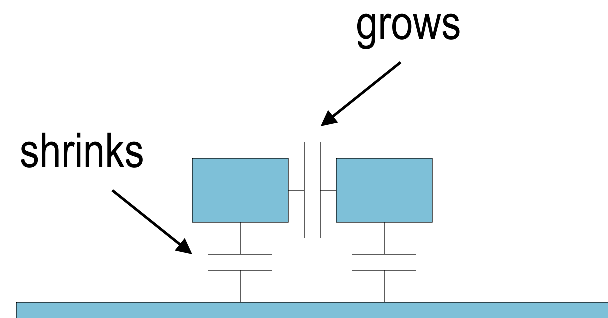


Wire delay ~ RC

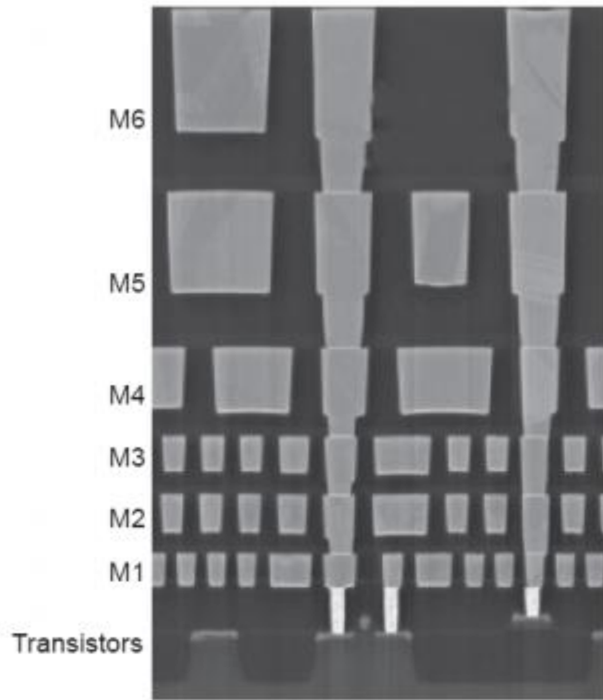
$$RC = \frac{\rho L}{t_m w} \frac{\varepsilon w L}{t_i} = \rho \varepsilon \frac{L^2}{t_m t_i}$$



scale

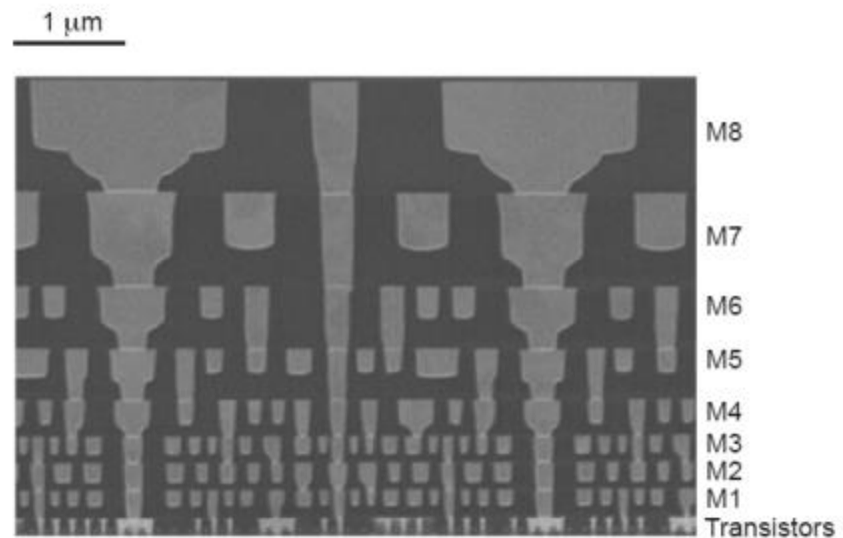


Further reading: https://asic-soc.blogspot.com/2008/10/net-delay.html



Intel 90 nm Stack

[Thompson02]



Intel 45 nm Stack

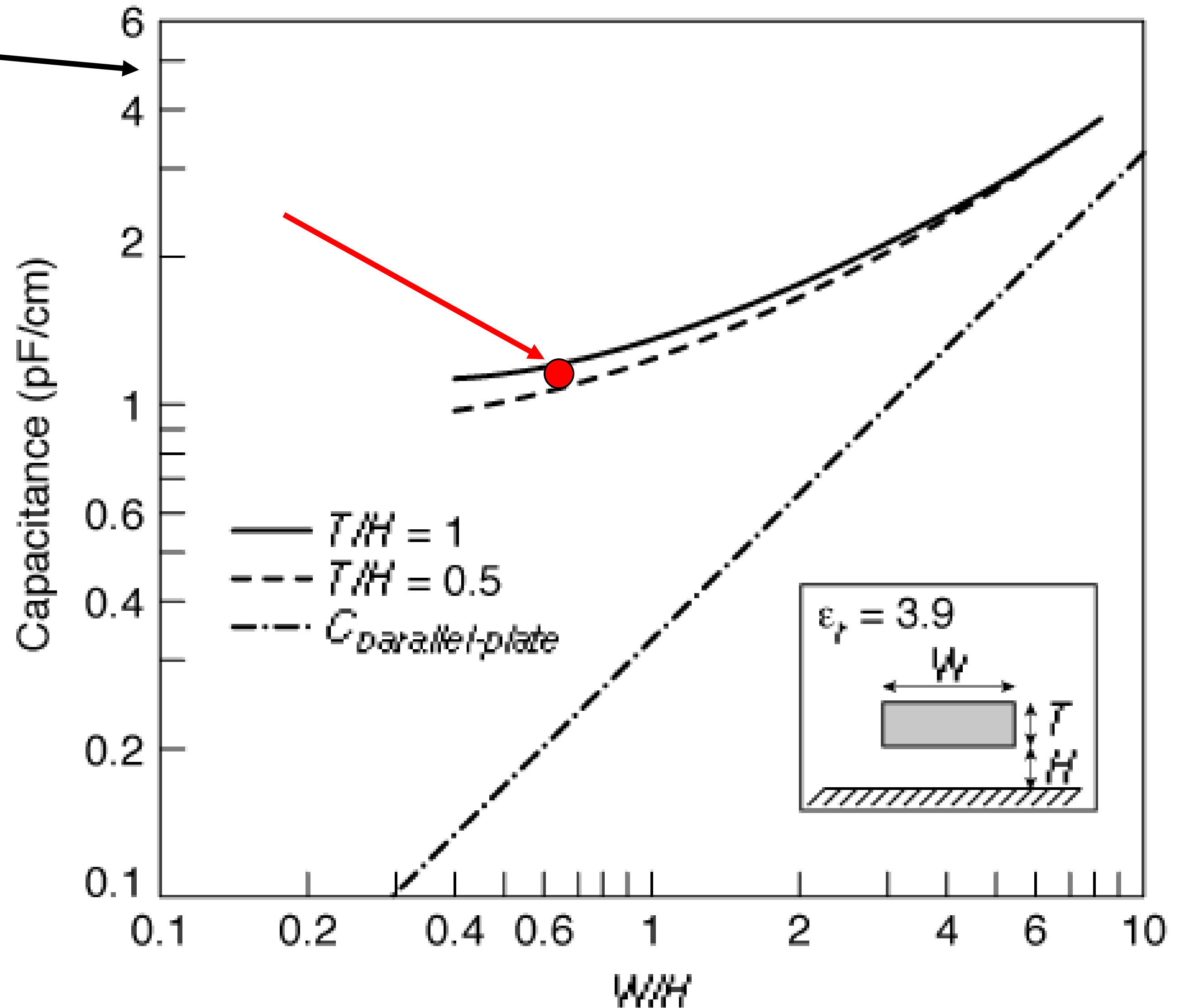
[Moon08]

But reality is worse…. (Fringe)

(from [Bakoglu89])

For Intel 0.25u process

- W~=0.64
- T~=0.48
- H is around 0.9.



www.ee.bgu.ac.il/~Orly_lab/courses/Intro_Course/Slides/Lecture02-2-Wire.ppt

And reducing the number of instructions executed…

- Sorry, wrong class.
  - Compilers can help with this (a lot in some cases)
  - So can ISA design, but making instructions too complex hurts ILP and t_{CLK}
    - Not clear there is a lot of room here for improvement.
- So on the whole reducing # of instructions doesn't look to be viable.
  - So ILP would seem to be "where it's at"

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)

Superscalar Out-of-Order

SuperScalar OoO

- Out-of-order and superscalar make for a nice combination
  - The whole point of OoO is to find something to do.
    - Superscalar provides the resources to **do** it.
- Out-of-order scales pretty nicely
  - Dependencies resolved at rename
  - True dependencies dealt with already by rename and the general OoO model.
- So we’ve already done a lot of the work.

But more to go

- To be superscalar one needs to be ***able*** to complete more than 1 instruction per cycle in a sustained way.
  - This means fetch, rename, issue, execute, CDB broadcast and retire must all be able to do 2 instructions at once.
  - It is mostly a matter of scale.

Fetch

- Performing more than one fetch seems straightforward.
  - Just grab PC and PC+4
  - It can be complicated by hardware restrictions
    - Say the two instructions span a cacheline
  - Branches also cause problems
    - What if PC+4 is wrong?
  - But as long as you can usually/often fetch two life is good.
    - And we can add tricks to handle these problems
      - Trace cache, multi-branch predictor, Icache annotations

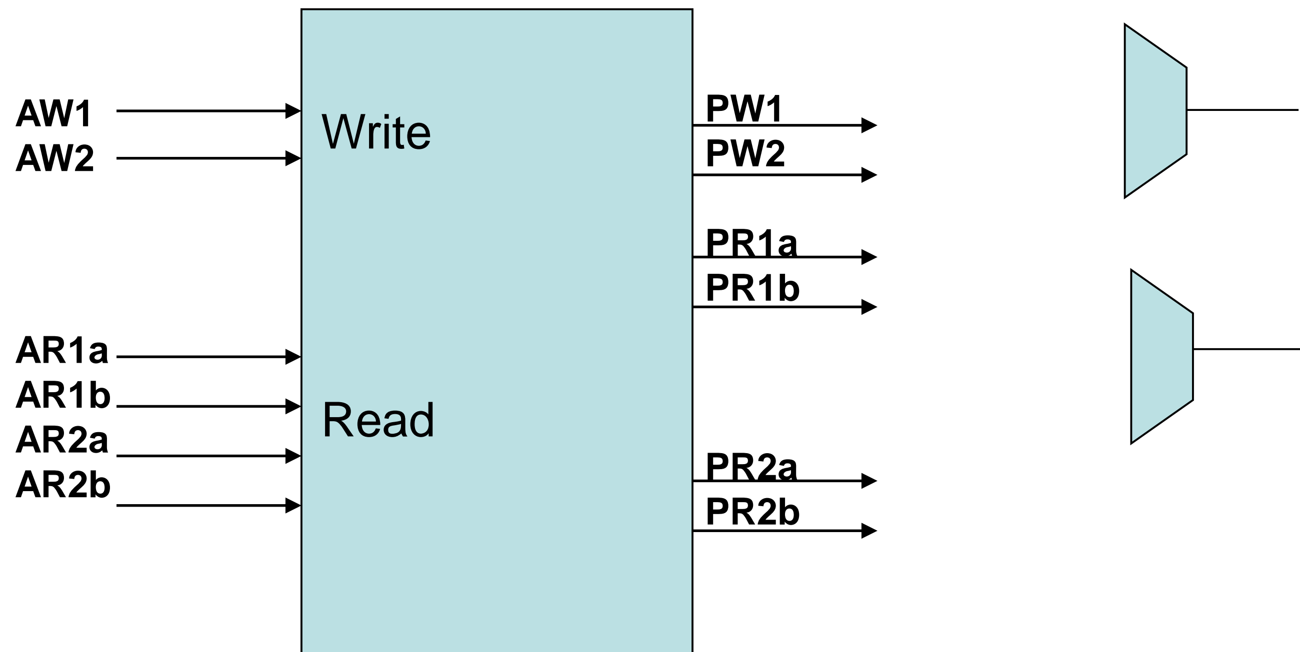
Decode

- Just have two of them.
  - For x86 or other CISC this might be unreasonable
    - Trace cache or otherwise caching decoded instructions might help here.

Rename

- One change is we need more ports to the RAT.
- Other (bigger) issue is making sure dependencies inside of the group are dealt with.
    - R1=…
    - …=R1
  - How do we handle this?
    - Basically similar to “register forwarding” inside of the register file

RAT



Situation: Two instructions (1 and 2) come in to the RAT. RAT renames two source registers per instruction (ARXa and ARXb) and allocates new PRF for two destination locations.

Dispatch

- Need to be able to send more than one instruction to the RoB and RS per cycle
  - Just more ports in RS and RoB.
  - A bit more complexity with “Ships passing in the Night”.
  - Read ports in PRF (R10K)
    - Read ports in ARF/ROB (P6)

Wake-up/select

- We’ve already been allowing more than one instruction to go to an exec unit per cycle.
  - No real change here.

Execute complete/CDB

- As two instructions need to be able to complete per cycle need more than one CDB.
  - In general everyone has to listen to everyone
  - Could try to partition CDBs but this is tricky.
  - Makes RS's bigger and probably slower.
- RoB needs ***yet more*** ports.

Commit

- In R10K this isn't a big deal.
  - But need to realize that more than one instruction at the head of the RoB is done (more ports) and must be able to complete them (maybe more ports)
  - In P6, you've got to do more copies.
    - Multiple read ports (RoB); multiple write ports (ARF)

LSQ

LSQ issues

- Load/Store queue
  - It is pretty tricky to get right.
    - Send to LSQ at issue
    - Does this replace the RS?
      - Maybe…
  - Probably a Store (circular) queue and a load buffer
    - Loads need to note which stores they care about
      - Ones that were there when the load issued
      - Need to not get caught by “wrap around” of the store queue
    - Loads need to check for what exactly?

So what to do?

- You have a lot of options on load launch
  - Conservative
    - Launch loads at the head of the LSQ (bad)
  - Moderate
    - Launch loads at when no conflicting/unknown stores exist in front of you (ok)
  - Aggressive
    - Launch loads ASAP, but fix if wrong.
    - Lots of potential issues.
      - Imagine you launched a load then solve it by forwarding. What happens when the load returns?
- And store forwarding might be tricky.
  - Can you identify the situation when you can forward?
    - If so, can you write verilog code for that?

Non-LSQ options

- Just launch loads from the RoB when they hit the head (easy/poor)
- As above, but prefetch the data into the cache ASAP.
  - This might actually work well. Probably need non-direct-mapped cache though.
- Use RoB to track when load has no conflicting/unknown stores in front of it.
  - Seems annoying, might be easy. Still poorish performance.

More on memory disambiguation later

Misc other stuff

More details

- RS
  - We’ve been doing generic RSs
  - Could also dedicate groups of RSs to a single execution unit (or group of similar execution units).
    - May make the RSs simpler
    - Will result in needing more total RSs to get the same performance
    - Everyone needs to listen to the CDB
    - For the project, means you have a bunch of similar code. Often a bad idea when fixing bugs.

Reading the register file on the way to EX

- The idea is to not read the PRF or RoB or CDB for the ***value***, only for the fact that the value is available.
    - Grab the value on your way to the EX unit.
  - Advantages
    - No CDB broadcast of values
    - Don't need to look in the PRF/ARF/RoB for values on issue.
  - Disadvantages
    - Already slow Wake-up/select/dispatch now slower
      - (But as we may be pipelining this anyways, not too bad).

Back-to-Back dependent instructions

- What has to happen to get them to go back to back?
  - Why is this hard?
  - How is it solved in the real world?

Looking back (and foward)

Looking back

- Keep the big picture in mind
  - OoO is about finding ILP.
      - If you don't need ILP don't bother
    - Why might you not need ILP?
      - Application doesn't need it
      - TLP instead.
  - In many ways this is about finding work to do while high-latency instructions run
    - If you fix the memory problem, it isn't clear that OoO makes any sense.

Looking back

- Renaming is distinct from OoO
  - You can rename without OoO
    - (Not obviously useful!)
  - You can have OoO without renaming
    - In fact the first OoO processor, CDC 6600 used scoreboarding which had no renaming but is out-of-order.

Static vs. Dynamic reordering

- Some reordering is better done statically
  - Have a global view
    - (infinite window)
  - Have access to the original source
    - May tell you about ***intent*** or even ***behavior***.
      - Array behavior may make load/store conflicts easy to identify.
  - Regular code structures may lend themselves to optimal ordering
    - Software pipelining
- Just a one-time compile cost.
  - Saves power and hardware cost if reordering done in software!

Static vs. Dynamic

- Some things are better done dynamically
  - Have live data
    - Worst case vs. actual case
      - Load/Store ordering possible to get right without being paranoid about the worst case.
    - Program behavior may change based on data set
      - Branch prediction in particular
  - Can speculate
    - Static specifies program behavior. Much harder to speculate in the complier.

Looking forward

- There is a LOT more to architecture than out-of-order execution
  - Memory
    - If OoO is mostly about reordering around high-latency loads memory sounds important
  - Power
    - Modern processors are eating huge amounts of power and we can't cool them. So what can an architect do?
  - Multi-processors
    - One way to get performance is to have many processors working on a task.
  - Static reordering
    - As noted, saves power over dynamic & might be able to use both together to get a nice impact.