

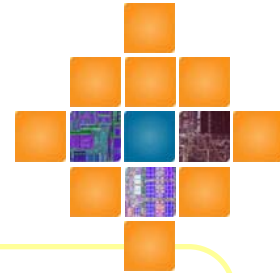
CS6290

Speculation Recovery

**Georgia
Tech**

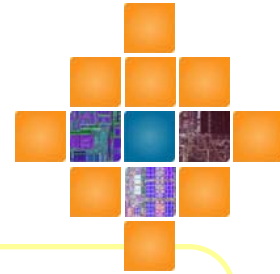


College of
Computing



Loose Ends

- Up to now:
 - Techniques for handling register dependencies
 - Register renaming for WAR, WAW
 - Tomasulo's algorithm for scheduling RAW
 - Branch prediction for control dependencies
- Still need to address:
 - Memory dependencies
 - Mispredictions, Faults, Exceptions



Speculative Fetch/Non-Spec Exec

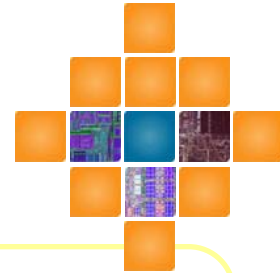
- Fetch instructions, even fetch past a branch, but don't exec right away



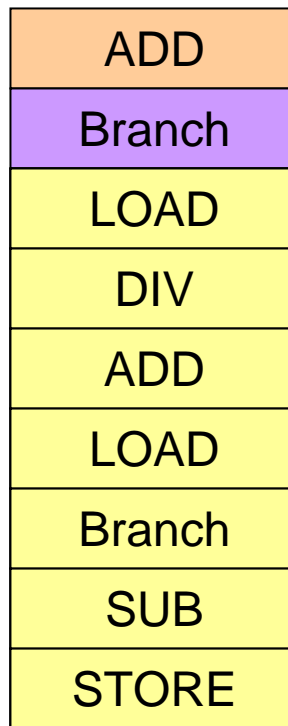
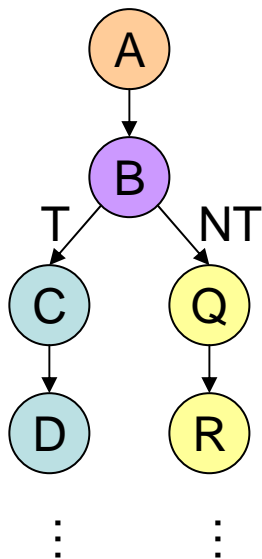
1. DIV R1 = R2 / R3
2. ADD R4 = R5 + R6
3. BEQZ R1, foo
4. SUB R5 = R4 - R3
5. MUL R7 = R5 * R2
6. MUL R8 = R7 * R3

	I	E	W	C

Branch Prediction/Speculative Execution



- When we hit a branch, guess if it's T or NT



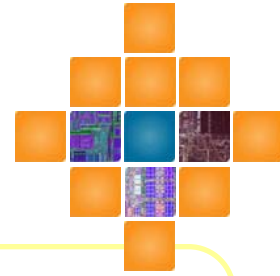
→ Guess T

Keep scheduling and executing instructions as we knew the outcome was T

Sometime later, if we messed up...

Just throw it all out

And fetch the correct instructions



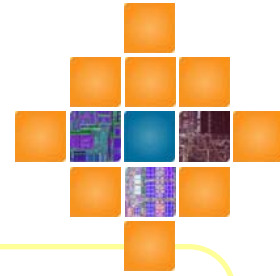
Again, with Speculative Execution

- Assume fetched direction is correct

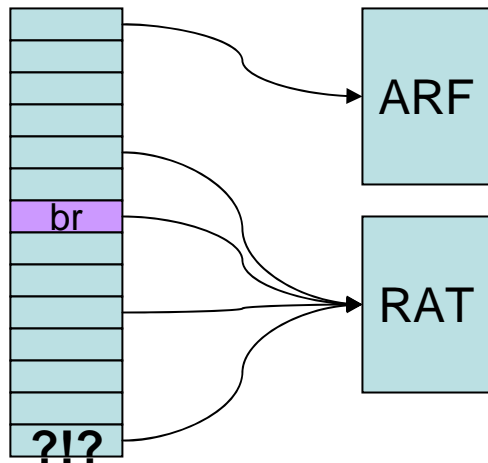


1. DIV R1 = R2 / R3
2. ADD R4 = R5 + R6
3. BEQZ R1, foo
4. SUB R5 = R4 - R3
5. MUL R7 = R5 * R2
6. MUL R8 = R7 * R3

	I	E	W	C



Branch Misprediction Recovery

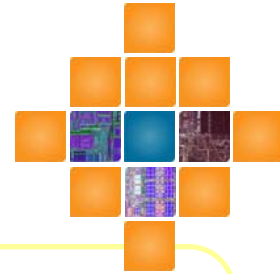


ARF state corresponds to state prior to oldest non-committed instruction

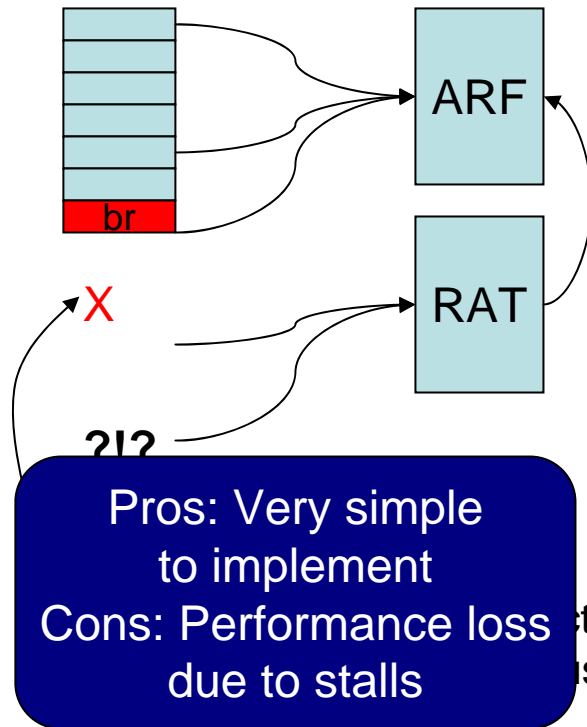
As instructions are processed, the RAT corresponds to the register mapping after the *most recently renamed* instruction

On a branch misprediction, wrong-path instructions are flushed from the machine

The RAT is left with an invalid set of mappings corresponding to the wrong-path instruction state



Solution 1: Stall and Drain

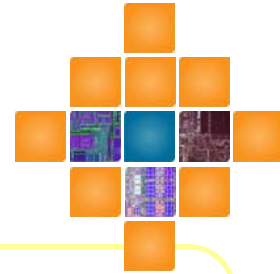


Allow all instructions to execute and commit; ARF corresponds to last committed instruction

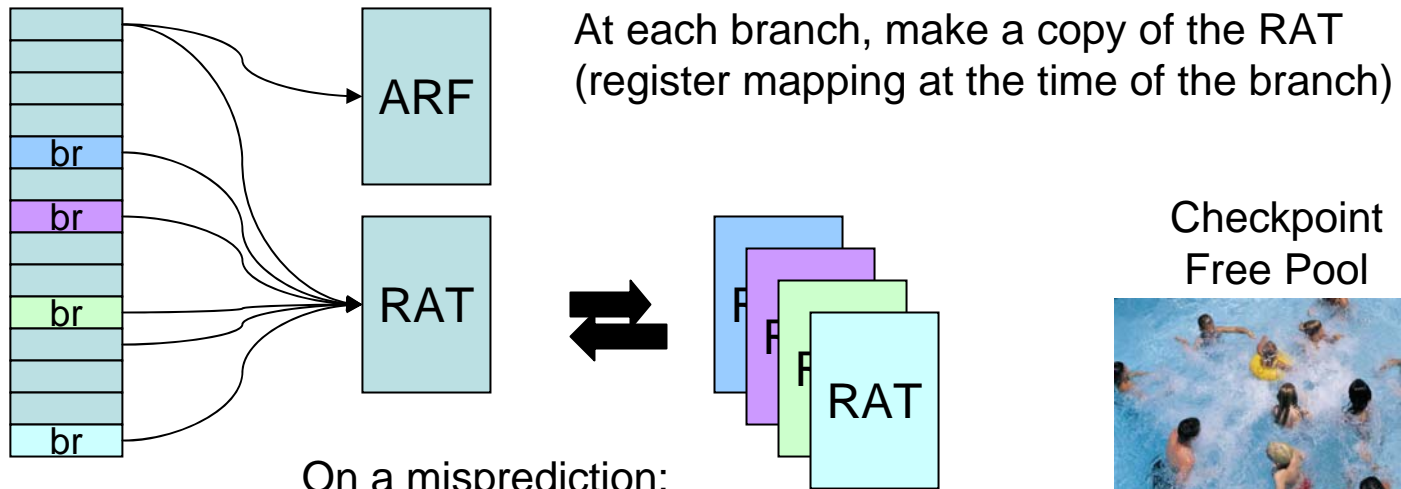
ARF now corresponds to the state right before the next instruction to be renamed (foo)

Reset RAT so that all mappings refer to the ARF

Resume renaming the new correct- path instructions from fetch
since RAT is wrong



Solution 2: Checkpointing

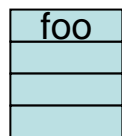


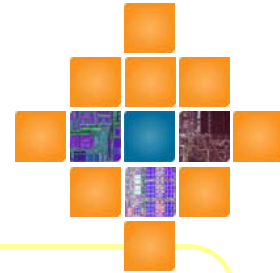
Checkpoint
Free Pool



On a misprediction:

1. flush wrong-path instructions
2. deallocate RAT checkpoints
3. recover RAT from checkpoint
4. resume renaming



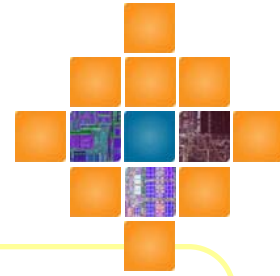


Speculative Execution is OK

- ROB maintains program order
- ROB temporarily stores results
 - If we screw something up, only the ROB knows, but no architected state is affected
- Register rename recovery makes sure we resume with correct register mapping

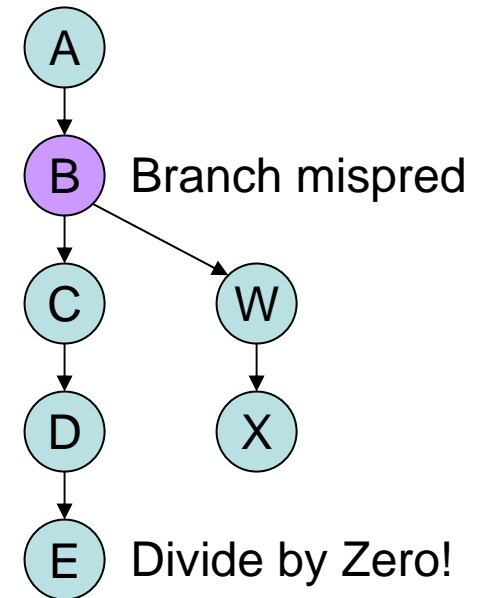
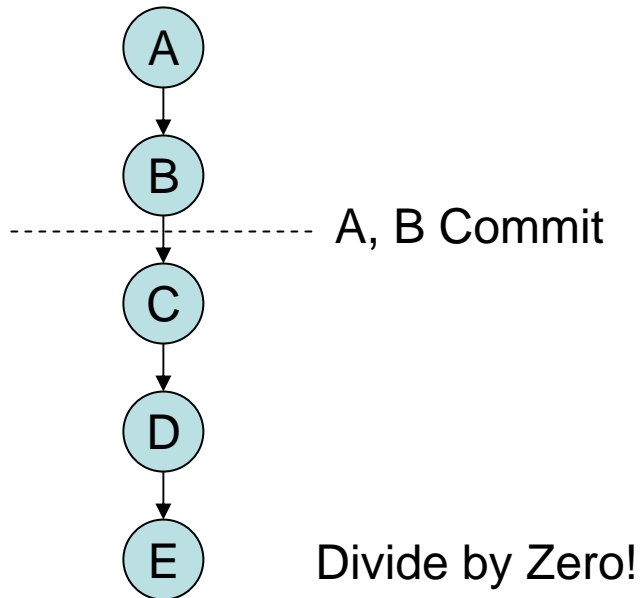
If we mess up, we:

1. Can undo the effects
2. Know how to resume



Commit, Exceptions

- What happens if a speculatively executed instruction faults?

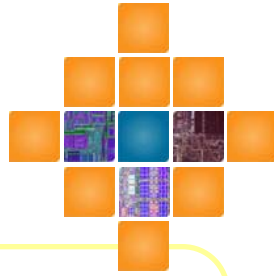


Outside world sees: A, B, fault!

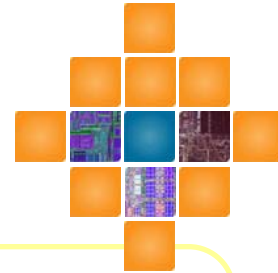
Should have been: A, B, C, D, fault!

Fault should never have happened!

Treat Fault Like a Result



- Regular register results written to ROB until
 - Instruction is oldest for in-order state update
 - Instruction is known to be non-speculative
- Do the same with faults!
 - At exec, make note of any faults, but don't expose to outside world
 - If the instruction gets to commit, then expose the fault



Example

A LOAD R1 = 0[R2] (commit)

B ADD R3 = R1 + R4

C SUB R1 = R5 - R6

D DIV R4 = R7 / R1

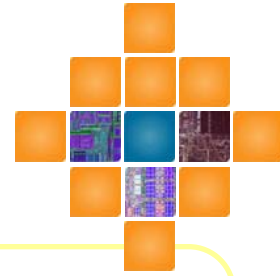
E LOAD R6 = 0[R7] (3 commits)

Resolved

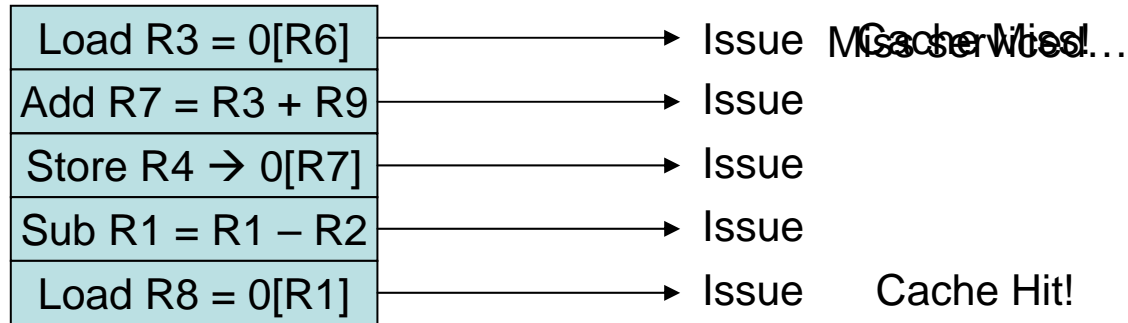
				E	C	F
LOAD	P1	imm	R2	X	X	
ADD	P2	P1	R4	X	X	
SUB	P3	R5	R6	X	X	
DIV	P4	R7	P3	X	X	X
LOAD	P5	imm	R7	X	X	X

Flush rest of ROB,
Start fetching
Fault handler

Fault!
Divide by zero
Fault deferred until arch
Now raise fault
Other fault "never happened"...

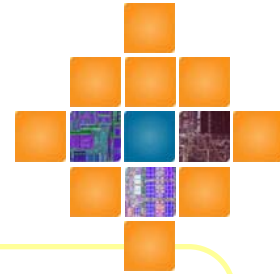


Executing Memory Instructions



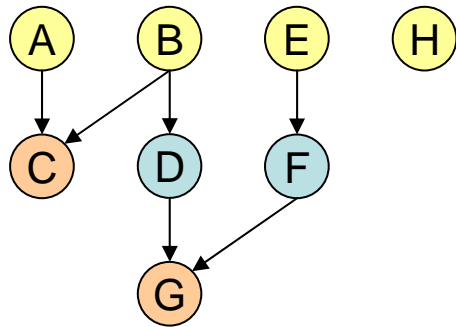
But there was a later load...

- If $R1 \neq R7$, then Load R8 gets correct value from cache
- If $R1 == R7$, then Load R8 should have gotten value from the Store, *but it didn't!*

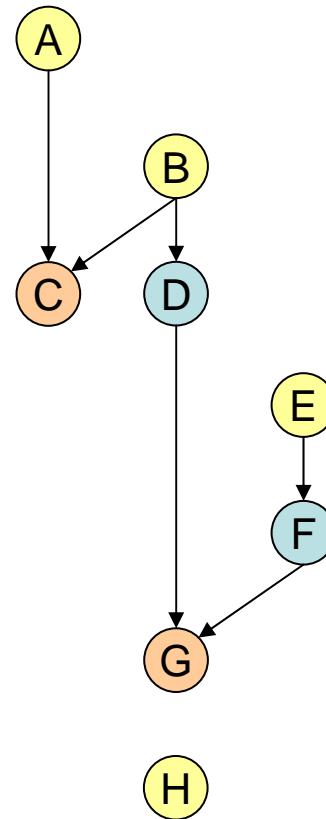


Out-of-Order Load Execution

- So don't let loads execute out-of-order!



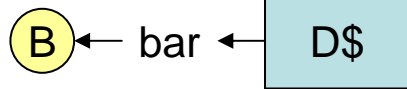
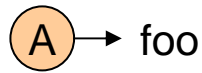
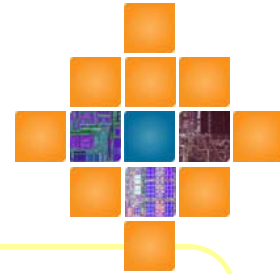
$$\text{ILP} = 8/3 = 2.67$$



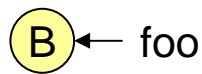
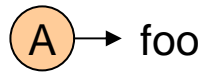
$$\text{ILP} = 8/7 = 1.14$$

Ok, maybe not a good idea. No support for OOO load execution can reduce your ILP

What Else Could Happen?

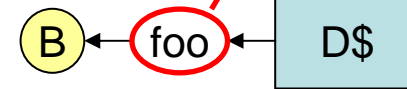
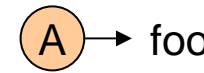


No problem.

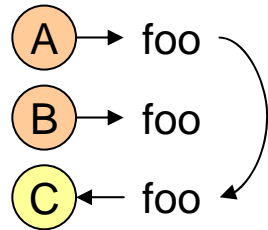


No problem.

Some sort of data forwarding mechanism

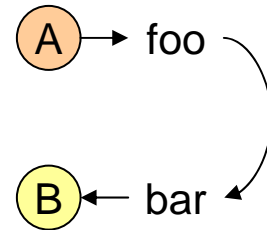


Uh oh.



Should have used B's store value

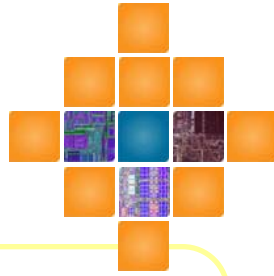
Uh oh.



Luckily, this usually can't even happen

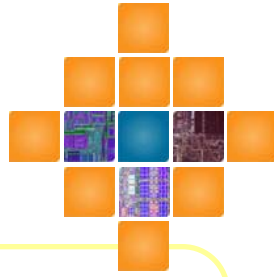
Uh oh.

Memory Disambiguation Problem

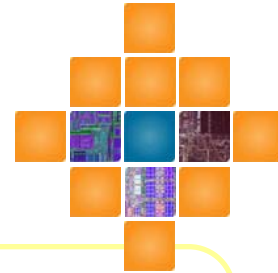


- Why can't this happen with non-memory insts?
 - Operand specifiers in non-memory insts are absolute
 - “R1” refers to one specific location
 - Operand specifiers in memory insts are ambiguous
 - “R1” refers to a memory location specified by the value of R1. As pointers change, so does this location.

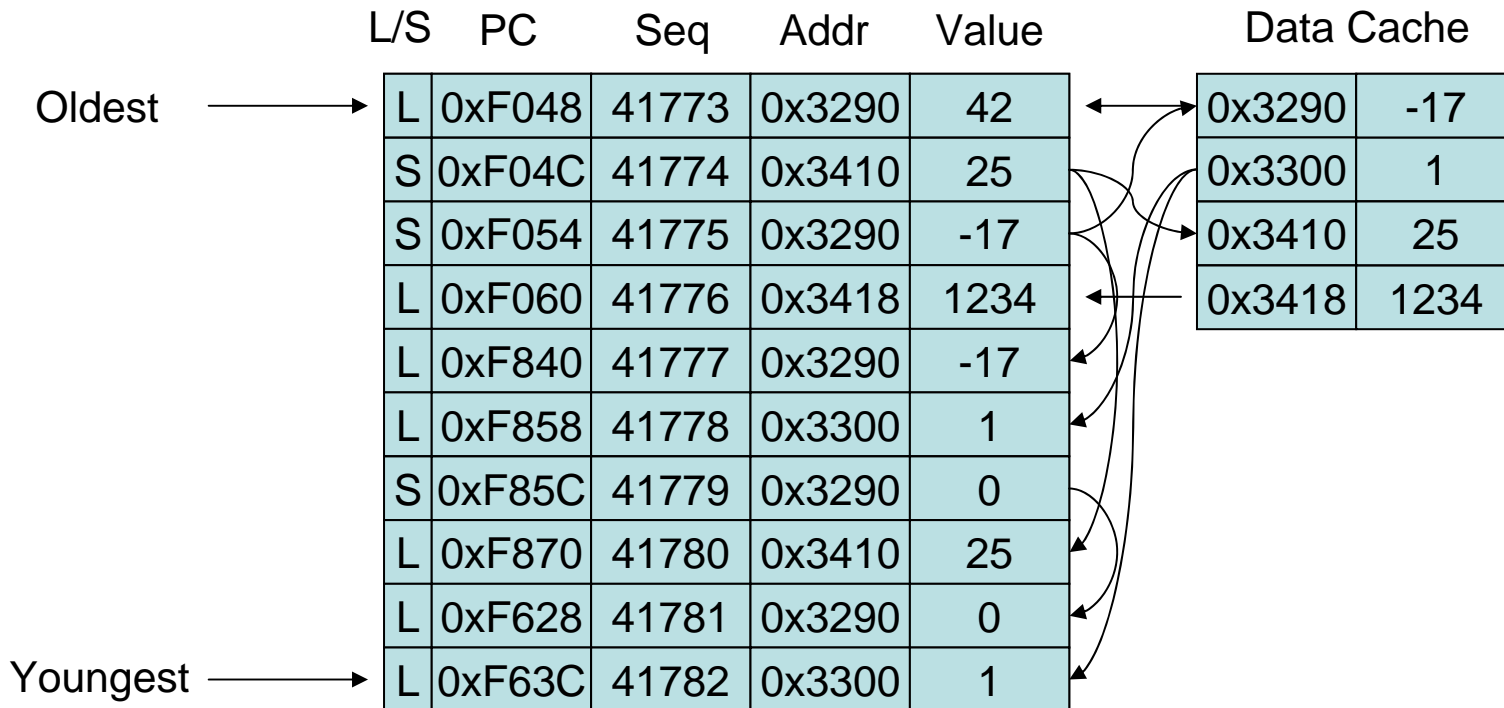
Two Problems



- Memory disambiguation
 - Are there any earlier unexecuted stores to the same address as myself? (I'm a load)
 - Binary question: answer is yes or no
- Store-to-load forwarding problem
 - Which earlier store do I get my value from? (I'm a load)
 - Which later load(s) do I forward my value to? (I'm a store)
 - Non-binary question: answer is one or more instruction identifiers



Load Store Queue (LSQ)



Disambiguation: loads cannot execute until all earlier store addresses computed

Forwarding: broadcast/search entire LSQ for match