HANDLING MEMORY OPS

Dynamically Scheduling Memory Ops

- Compilers must schedule memory ops conservatively
- Options for hardware:
  - Hold loads until all prior stores execute (conservative)
  - Execute loads as soon as possible, detect violations (aggressive)
    - When a store executes, it checks if any later loads executed too early (to same address). If so, flush pipeline
  - Learn violations over time, selectively reorder (predictive)

Before

```
ld r2,4(sp)
ld r3,8(sp)
add r3,r2,r1   //stall
st r1,0(sp)
ld r5,0(r8)
ld r6,4(r8)
sub r5,r6,r4   //stall
st r4,8(r8)
```

Wrong(?)

```
ld r2,4(sp)
ld r3,8(sp)
ld r5,0(r8) //does r8==sp?
add r3,r2,r1
ld r6,4(r8) //does r8+4==sp?
st r1,0(sp)
sub r5,r6,r4
st r4,8(r8)
```

Loads and Stores

Instruction	Disp	Issue	WB	Commit
`fdiv p1,p2 → p3`	1	2	25	
`st p4 → [p5]`	1	2	3	
`st p3 → [p6]`	2			
`ld [p7] → p8`	2			

Cycle 3:

- Can ld [p7]→p8 execute? (why or why not?)

Loads and Stores

Instruction	Disp	Issue	WB	Commit
`fdiv p1,p2 → p3`	1	2	25	
`st p4 → [p5]`	1	2	3	
`st p3 → [p6]`	2			
`ld [p7] → p8`	2			

Aliasing (again)

- p5 == p7 ?
- p6 == p7 ?

Loads and Stores

Instruction	Disp	Issue	WB	Commit
fdiv p1,p2 → p3	1	2	25	
st p4 → [p5]	1	2	3	
st p3 → [p6]	2			
ld [p7] → p8	2			

Suppose p5 == p7 and p6 != p7

- Can ld [p7]→p8 execute? (why or why not?)

Memory Forwarding

- Stores write cache at commit
  - Commit is in-order, delayed by all instructions
  - Allows stores to be "undone" on branch mis-predictions, etc.
- Loads read cache
  - Early execution of loads is critical
- Forwarding
  - Allow store → load communication before store commit
  - Conceptually like reg. bypassing, but different implementation
    - Why? Addresses unknown until execute

Forwarding: Store Queue

Store Queue

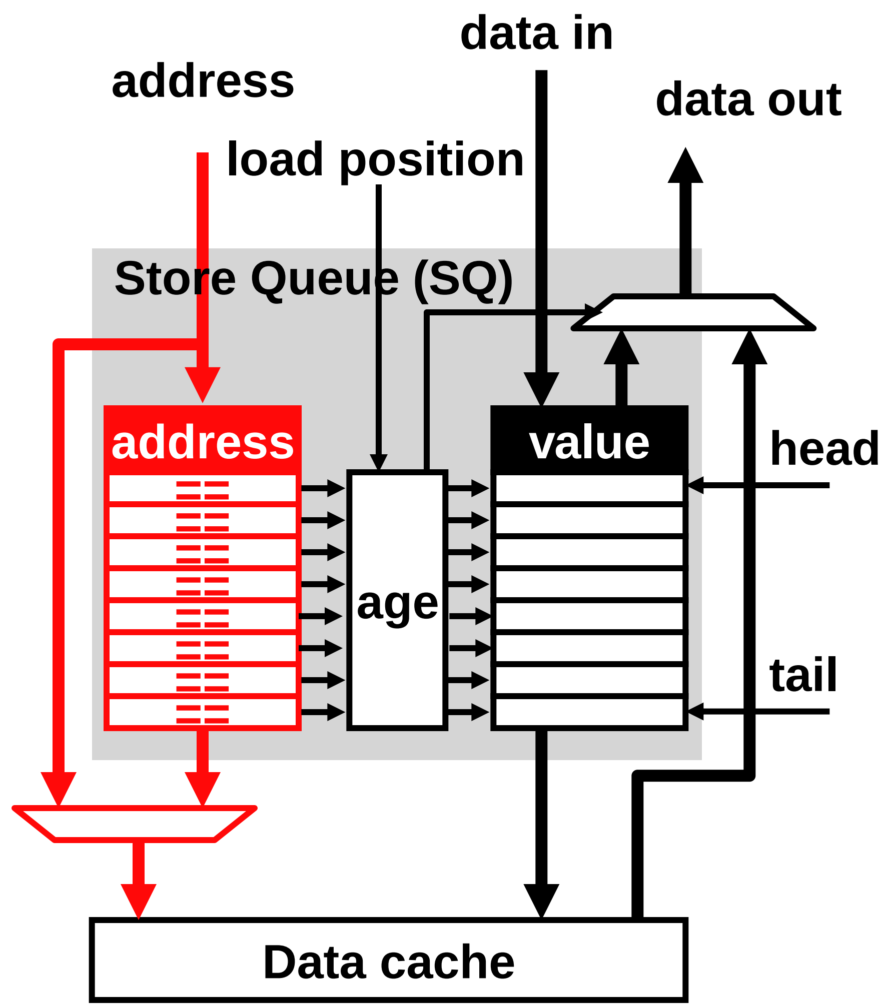
- Holds all in-flight stores
- CAM: searchable by address
- Age logic: determine youngest matching store older than load

Store execution

- Write Store Queue
  - Address + Data

Load execution

- Search SQ
  - Match? Forward
- Read D$



Load scheduling

- Store→Load Forwarding:
  - Get value from executed (but not comitted) store to load
- Load Scheduling:
  - Determine when load can execute with regard to older stores

- Conservative load scheduling:
  - All older stores have executed
  - Some architectures: split store address / store data
    - Only require known address
  - Advantage: always safe
  - Disadvantage: performance (limits out-of-orderness)

Our example from before

```
ld   [r1] → r5
ld   [r2] → r6
add r5,r6 → r7
st   r7 → [r3]
ld   4[r1] → r5
ld   4[r2] → r6
add r5,r6 → r7
st   r7 → 4[r3]
// loop control here
```

With conservative load scheduling, what can go out of order?

Our example from before

		Disp	Issue	WB	Commit
1	`ld [p1] → p5`	1			
2	`ld [p2] → p6`	1			
3	`add p5,p6 → p7`				
4	`st p7 → [p3]`				
5	`ld 4[p1] → p8`				
6	`ld 4[p2] → p9`				
7	`add p8,p9 → p4`				
8	`st p4 → 4[p3]`				

- 2 wide, conservative scheduling
- issue 1 load per cycle
- loads take 3 cycles to complete

Cycle 1:
Dispatch insns #1, #2

Our example from before

		Disp	Issue	WB	Commit
1	`ld [p1] → p5`	1	2	5	
2	`ld [p2] → p6`	1			
3	`add p5,p6 → p7`	2			
4	`st p7 → [p3]`	2			
5	`ld 4[p1] → p8`				
6	`ld 4[p2] → p9`				
7	`add p8,p9 → p4`				
8	`st p4 → 4[p3]`				

- 2 wide, conservative scheduling
- issue 1 load per cycle
- loads take 3 cycles to complete

Cycle 2:
Why don’t we issue #2?

Our example from before

		Disp	Issue	WB	Commit
1	`ld [p1] → p5`	1	2	5	
2	`ld [p2] → p6`	1	3	6	
3	`add p5,p6 → p7`	2			
4	`st p7 → [p3]`	2			
5	`ld 4[p1] → p8`	3			
6	`ld 4[p2] → p9`	3			
7	`add p8,p9 → p4`				
8	`st p4 → 4[p3]`				

- 2 wide, conservative scheduling
- issue 1 load per cycle
- loads take 3 cycles to complete

Cycle 3:
Why don't we issue #3?
Why don't we issue #4?

Our example from before

		Disp	Issue	WB	Commit
1	`ld [p1] → p5`	1	2	5	
2	`ld [p2] → p6`	1	3	6	
3	`add p5,p6 → p7`	2			
4	`st p7 → [p3]`	2			
5	`ld 4[p1] → p8`	3			
6	`ld 4[p2] → p9`	3			
7	`add p8,p9 → p4`	4			
8	`st p4 → 4[p3]`	4			

- 2 wide, conservative scheduling
- issue 1 load per cycle
- loads take 3 cycles to complete

Cycle 4:
Why don't we issue #5?

Our example from before

		Disp	Issue	WB	Commit
1	`ld [p1] → p5`	1	2	5	6
2	`ld [p2] → p6`	1	3	6	
3	`add p5,p6 → p7`	2	6	7	
4	`st p7 → [p3]`	2			
5	`ld 4[p1] → p8`	3			
6	`ld 4[p2] → p9`	3			
7	`add p8,p9 → p4`	4			
8	`st p4 → 4[p3]`	4			

- 2 wide, conservative scheduling
- issue 1 load per cycle
- loads take 3 cycles to complete

Cycle 6:
Finally some action!

Our example from before

		Disp	Issue	WB	Commit
1	`ld [p1] → p5`	1	2	5	6
2	`ld [p2] → p6`	1	3	6	7
3	`add p5,p6 → p7`	2	6	7	
4	`st p7 → [p3]`	2	7	8	
5	`ld 4[p1] → p8`	3			
6	`ld 4[p2] → p9`	3			
7	`add p8,p9 → p4`	4			
8	`st p4 → 4[p3]`	4			

- 2 wide, conservative scheduling
- issue 1 load per cycle
- loads take 3 cycles to complete

Cycle 7:
Getting somewhere….

Our example from before

		Disp	Issue	WB	Commit
1	`ld [p1] → p5`	1	2	5	6
2	`ld [p2] → p6`	1	3	6	7
3	`add p5,p6 → p7`	2	6	7	8
4	`st p7 → [p3]`	2	7	8	
5	`ld 4[p1] → p8`	3	8	11	
6	`ld 4[p2] → p9`	3			
7	`add p8,p9 → p4`	4			
8	`st p4 → 4[p3]`	4			

- 2 wide, conservative scheduling
- issue 1 load per cycle
- loads take 3 cycles to complete

Cycle 8:
Etc…

Our example from before

		Disp	Issue	WB	Commit
1	`ld [p1] → p5`	1	2	5	6
2	`ld [p2] → p6`	1	3	6	7
3	`add p5,p6 → p7`	2	6	7	8
4	`st p7 → [p3]`	2	7	8	9
5	`ld 4[p1] → p8`	3	8	11	
6	`ld 4[p2] → p9`	3	9	12	
7	`add p8,p9 → p4`	4			
8	`st p4 → 4[p3]`	4			

- 2 wide, conservative scheduling
- issue 1 load per cycle
- loads take 3 cycles to complete

Cycle 9:
Etc…

Our example from before

		Disp	Issue	WB	Commit
1	`ld [p1] → p5`	1	2	5	6
2	`ld [p2] → p6`	1	3	6	7
3	`add p5,p6 → p7`	2	6	7	8
4	`st p7 → [p3]`	2	7	8	9
5	`ld 4[p1] → p8`	3	8	11	12
6	`ld 4[p2] → p9`	3	9	12	
7	`add p8,p9 → p4`	4	12	13	
8	`st p4 → 4[p3]`	4			

- 2 wide, conservative scheduling
- issue 1 load per cycle
- loads take 3 cycles to complete

Cycle 12:
Yawn…

Our example from before

		Disp	Issue	WB	Commit
1	`ld [p1] → p5`	1	2	5	6
2	`ld [p2] → p6`	1	3	6	7
3	`add p5,p6 → p7`	2	6	7	8
4	`st p7 → [p3]`	2	7	8	9
5	`ld 4[p1] → p8`	3	8	11	12
6	`ld 4[p2] → p9`	3	9	12	13
7	`add p8,p9 → p4`	4	12	13	
8	`st p4 → 4[p3]`	4	13	14	

- 2 wide, conservative scheduling
- issue 1 load per cycle
- loads take 3 cycles to complete

Cycle 13:
Stretch…

Our example from before

		Disp	Issue	WB	Commit
1	`ld [p1] → p5`	1	2	5	6
2	`ld [p2] → p6`	1	3	6	7
3	`add p5,p6 → p7`	2	6	7	8
4	`st p7 → [p3]`	2	7	8	9
5	`ld 4[p1] → p8`	3	8	11	12
6	`ld 4[p2] → p9`	3	9	12	13
7	`add p8,p9 → p4`	4	12	13	14
8	`st p4 → 4[p3]`	4	13	14	

- 2 wide, conservative scheduling
- issue 1 load per cycle
- loads take 3 cycles to complete

Cycle 14:
Zzzzzz…

Our example from before

		Disp	Issue	WB	Commit
1	`ld [p1] → p5`	1	2	5	6
2	`ld [p2] → p6`	1	3	6	7
3	`add p5,p6 → p7`	2	6	7	8
4	`st p7 → [p3]`	2	7	8	9
5	`ld 4[p1] → p8`	3	8	11	12
6	`ld 4[p2] → p9`	3	9	12	13
7	`add p8,p9 → p4`	4	12	13	14
8	`st p4 → 4[p3]`	4	13	14	15

- 2 wide, conservative scheduling
- issue 1 load per cycle
- loads take 3 cycles to complete

Cycle 15:
2-wide ooo = 1-wide inorder
I am going to cry.

Our example from before

		Disp	Issue	WB	Commit
1	`ld [p1] → p5`	1	2	5	6
2	`ld [p2] → p6`	1	3	6	7
3	`add p5,p6 → p7`	2	6	7	8
4	`st p7 → [p3]`	2	7	8	9
5	`ld 4[p1] → p8`	3	8	11	12
6	`ld 4[p2] → p9`	3	9	12	13
7	`add p8,p9 → p4`	4	12	13	14
8	`st p4 → 4[p3]`	4	13	14	15

- 2 wide, conservative scheduling
- issue 1 load per cycle
- loads take 3 cycles to complete

What was **#5** waiting for??

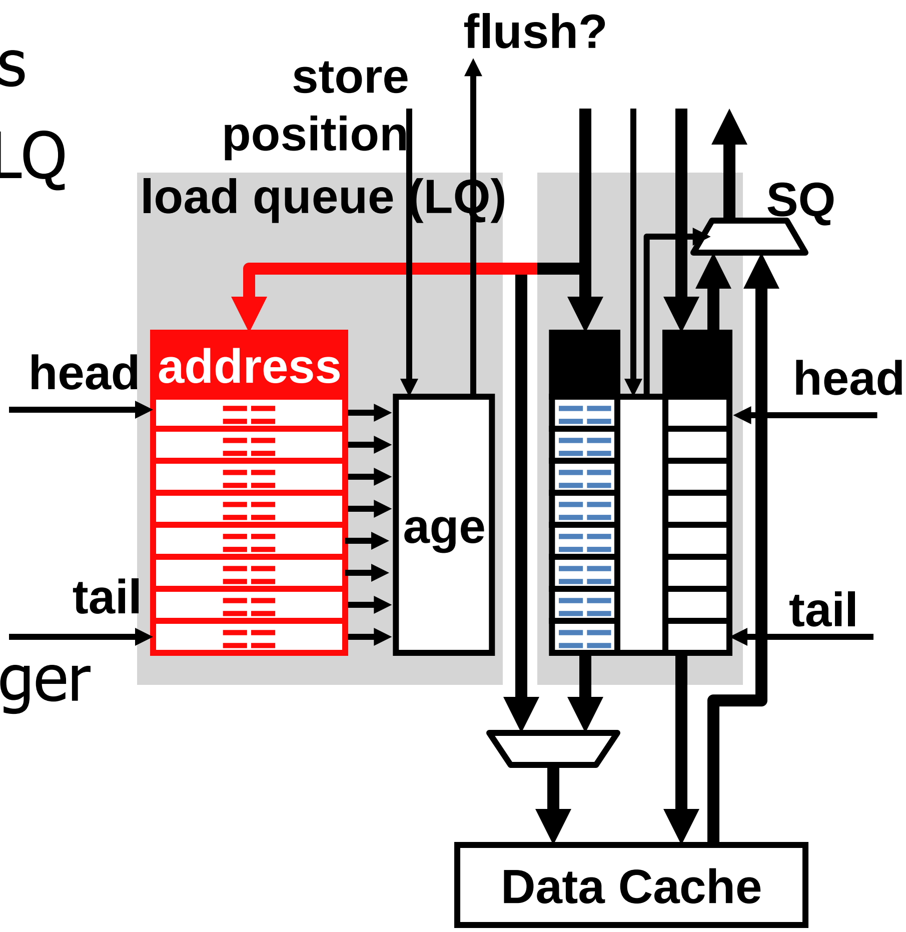
Can I speculate?

Load Speculation

- Speculation requires two things…..
  - Detection of mis-speculations
    - How can we do this?
  - Recovery from mis-speculations
    - Squash from offending load
    - Saw how to squash from branches: same method

Load Queue

- Detects ld ordering violations
- Execute load: write addr to LQ
  - Also note any store forwarded from
- Execute store: search LQ
  - Younger load with same addr?
  - Didn't forward from younger store?



Store Queue + Load Queue

- Store Queue: handles forwarding
  - Written by stores (@ execute)
  - Searched by loads (@ execute)
  - Read SQ when you write to the data cache (@ commit)

- Load Queue: detects ordering violations
  - Written by loads (@ execute)
  - Searched by stores (@ execute)

- Both together
  - Allows aggressive load scheduling
    - Stores don't constrain load execution

Our example from before

		Disp	Issue	WB	Commit
1	`ld [p1] → p5`	1	2	5	
2	`ld [p2] → p6`	1	3	6	
3	`add p5,p6 → p7`	2			
4	`st p7 → [p3]`	2			
5	`ld 4[p1] → p8`	3	4	7	
6	`ld 4[p2] → p9`	3			
7	`add p8,p9 → p4`	4			
8	`st p4 → 4[p3]`	4			

- 2 wide, **aggressive** scheduling
- issue 1 load per cycle
- loads take 3 cycles to complete

Cycle 4:
Speculatively execute #5 before the store (#4).

Our example from before

		Disp	Issue	WB	Commit
1	`ld [p1] → p5`	1	2	5	
2	`ld [p2] → p6`	1	3	6	
3	`add p5,p6 → p7`	2			
4	`st p7 → [p3]`	2			
5	`ld 4[p1] → p8`	3	4	7	
6	`ld 4[p2] → p9`	3	5	8	
7	`add p8,p9 → p4`	4			
8	`st p4 → 4[p3]`	4			

- 2 wide, **aggressive** scheduling
- issue 1 load per cycle
- loads take 3 cycles to complete

Cycle 5:
Speculatively execute #6 before the store (#4).

Our example from before

		Disp	Issue	WB	Commit
1	`ld [p1] → p5`	1	2	5	6
2	`ld [p2] → p6`	1	3	6	7
3	`add p5,p6 → p7`	2	6	7	8
4	`st p7 → [p3]`	2	7	8	9
5	`ld 4[p1] → p8`	3	4	7	9
6	`ld 4[p2] → p9`	3	5	8	10
7	`add p8,p9 → p4`	4	8	9	10
8	`st p4 → 4[p3]`	4	9	10	11

- 2 wide, **aggressive** scheduling
- issue 1 load per cycle
- loads take 3 cycles to complete

Fast forward:
4 cycles faster
Actually ooo this time!

Aggressive Load Scheduling

- Allows loads to issue before older stores
  - Increases out-of-orderness

  \+ When no conflict, increases performance

  \- Conflict → squash → worse performance than waiting

- Some loads might forward from stores
  - Always aggressive will squash a lot

- Can we have our cake AND eat it too?

Predictive Load Scheduling

- Predict which loads must wait for stores

- Fool me once, shame on you—fool me twice?
  - Loads default to aggressive
  - Keep table of load PCs that have been caused squashes
    - Schedule these conservatively

  \+ Simple predictor

  – Makes "bad" loads wait for *all* older stores: not great

- More complex predictors used in practice
  - Predict which stores loads should wait for

Out of Order: Window Size

- Scheduling scope = ooo window size
  - Larger = better
  - Constrained by physical registers (#preg)
    - ROB roughly limited by #preg = ROB size + #logical registers
    - Big register file = hard/slow
  - Constrained by issue queue
    - Limits number of un-executed instructions
    - CAM = can't make big (power + area)
  - Constrained by load + store queues
    - Limit number of loads/stores
    - CAMs
    - Active area of research: scaling window sizes
- Usefulness of large window: limited by branch prediction
  - 5% branch mis-prediction rate: 1 in 20 branches, 1 in 100 insns

Out of Order: Benefits

- Allows speculative re-ordering
  - Loads / stores
  - Branch prediction
- Schedule can change due to cache misses
  - Different schedule optimal from on cache hit
- Done by hardware
  - Compiler may want different schedule for different hw configs
  - Hardware has only its own configuration to deal with

Static vs. Dynamic Scheduling

- If we can do this in software…
- …why build complex (slow-clock, high-power) hardware?
  - \+ Performance portability
    - Don't want to recompile for new machines
  - \+ More information available
    - Memory addresses, branch directions, cache misses
  - \+ More registers available
    - Compiler may not have enough to schedule well
  - \+ Speculative memory operation re-ordering
    - Compiler must be conservative, hardware can speculate
  - – But compiler has a larger scope
    - Compiler does as much as it can (not much)
    - Hardware does the rest

Out of Order: Top 5 Things to Know

- Register renaming
  - How to perform it and how to recover it
- Commit
  - Precise state (ROB)
  - How/when registers are freed
- Issue/Select
  - Wakeup: CAM
  - Choose N oldest ready instructions
- Stores
  - Write at commit
  - Forward to loads via SQ
- Loads
  - Conservative/aggressive/predictive scheduling
  - Violation detection via LQ

Summary: Dynamic Scheduling

- Dynamic scheduling
  - Totally in the hardware
  - Also called “out-of-order execution” (OoO)
- Fetch many instructions into instruction window
  - Use branch prediction to speculate past (multiple) branches
  - Flush pipeline on branch misprediction
- Rename to avoid false dependencies
- Execute instructions as soon as possible
  - Register dependencies are known
  - Handling memory dependencies more tricky
- “Commit” instructions in order
  - Anything strange happens pre-commit, just flush the pipeline
- Current machines: 100+ instruction scheduling window