ECE 4750 Computer Architecture, Fall 2024

Topic 9: Advanced Processors Memory Disambiguation

Prof. Anne Bracy

Modified from slides developed by Drew Hilton (Duke University) and Milo Martin (Google)

Dynamically Scheduling Memory Insns

Options for hardware:

1. Hold loads until all prior stores execute (conservative)
2. 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

Before

```
a: LW x2,4(sp)
b: LW x3,8(sp)
c: ADD x1,x3,x2 //stalls
d: SW x1,0(sp)
e: LW x5,0(x8)
f: LW x6,4(x8)
g: SUB x4,x5,x6 //stalls
h: SW x4,8(x8)
```

Reorder to avoid stalls?

Improvement (?)

```
a: LW x2,4(sp)
b: LW x3,8(sp)
e: LW x5,0(x8)
c: ADD x1,x3,x2
f: LW x6,4(x8)
d: SW x1,0(sp)
g: SUB x4,x5,x6
h: SW x4,8(x8)
```

x8 not pending and it's good to give loads a head start

No one waits for a store, so good choice to fill the slot between f&g

Dynamically Scheduling Memory Insns

Options for hardware:

1. Hold loads until all prior stores execute (conservative)
2. 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

Before

```
a: LW x2,4(sp)
b: LW x3,8(sp)
c: ADD x1,x3,x2 //stalls
d: SW x1,0(sp)
e: LW x5,0(x8)
f: LW x6,4(x8)
g: SUB x4,x5,x6 //stalls
h: SW x4,8(x8)
```

possible RAW memory dependence?

Improvement (?)

Is this legal?

```
a: LW x2,4(sp)
b: LW x3,8(sp)
e: LW x5,0(x8) // x8==sp?
c: ADD x1,x3,x2
f: LW x6,4(x8) // r8+4==sp?
d: SW x1,0(sp)
g: sub x4,x5,x6
h: SW x4,8(x8)
```

Backwards arrows! ☹

Might not know at compile time. Cannot tell by inspecting register names.

Memory Forwarding

- Stores write cache at commit
  - Commit is in-order, delayed by all instructions
  - Allows stores to be "undone" on exceptions, branch mis-predictions, etc.

- Loads read cache
  - Early execution of loads is critical

- Forwarding
  - Allow store → load communication *before* commit
  - Conceptually like register bypassing, but different implementation
    - Why? Addresses unknown until execute

Forwarding: Store Queue

Store Queue

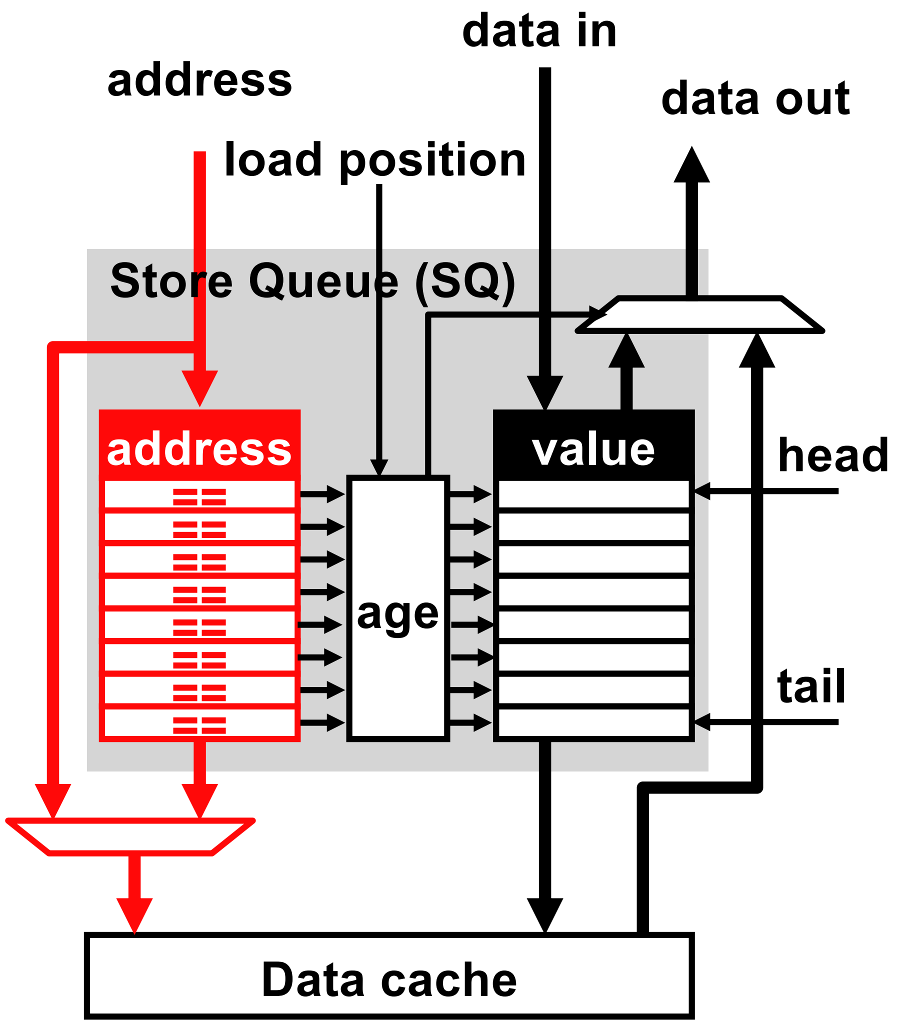
- Holds all in-flight stores
- searchable by address
- Age logic: determine youngest matching store older than load

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

Example: suppose ∃ is a RAW memory dependence between d & e

```
d: SW x1,0(sp)
e: LW x5,0(x8)
```

d:

- Writes the Store Queue @ Execute (address and value)
- Doesn’t write to the cache until commit

e:

- Checks the Store Queue @ Execute
  - sees address match between d & e
  - Value forwarded to e
    - just like register bypassing
    - e doesn't even need to go to the cache!

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

Example:

```
d: SW x1,0(sp)
e: LW x5,0(x8)
```

Suppose d hasn't even been issued yet (waiting on x1)

Do we let instruction e issue?

- What do we even know @ issue?

Conservative Load scheduling

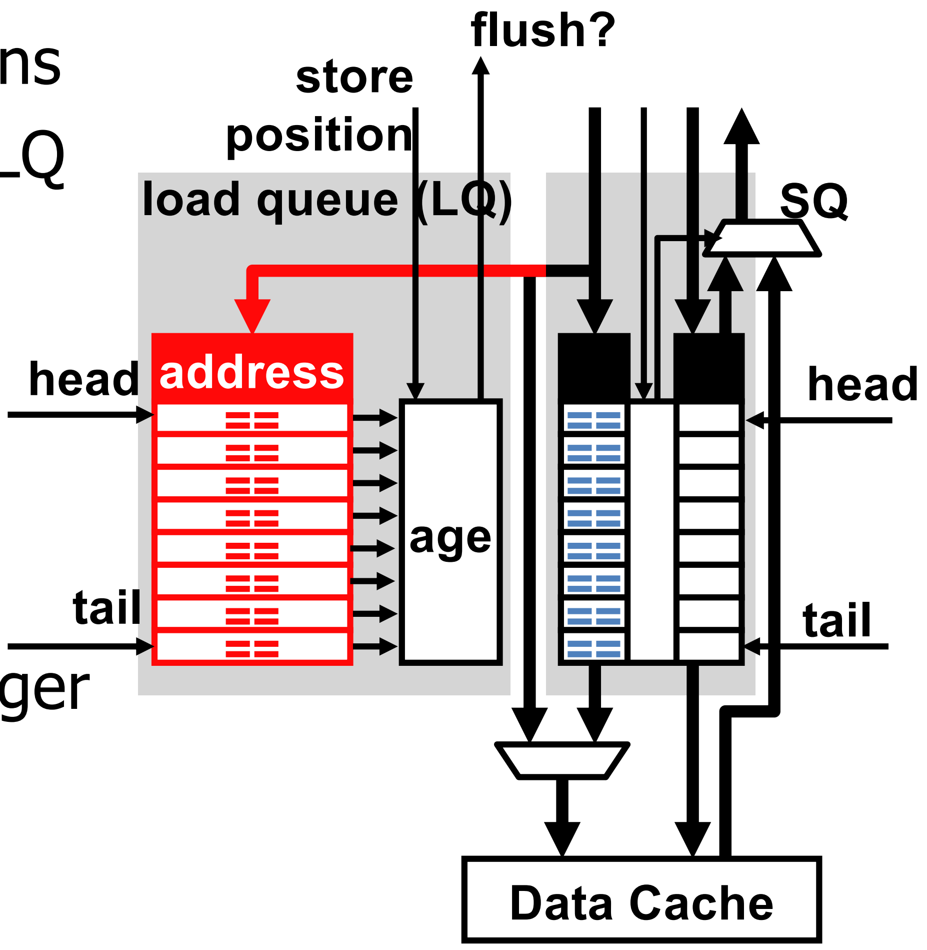
- Loads can only issue when all older stores have executed
- Some architectures: split store address / store data
  - Only require known address
- Advantage: always safe
- Disadvantage: performance (limits out-of-orderness)

Load Speculation

- Speculation requires two things…..
  - Detection of mis-speculations
    - How can we do this?
  - Recovery from mis-speculations
    - Squash from branches
      - Any instruction fetched after the mis-predicted branch gets squashed
    - Squash from offending load
      - Any instruction depending on the output of the load gets squashed

Load Queue

- Detects LW 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

Example (cycles 1-4)

		Decode	Issue	Complete	Commit
1	`LW x2,4(sp)`	1	2	5	
2	`LW x3,8(sp)`	1	3	6	
3	`ADD x1,x3,x2`	2			
4	`SW x1,0(sp)`	2			
5	`LW x5,0(x8)`	3	4	7	
6	`LW x6,4(x8)`	3			
7	`SUB x4,x5,x6`	4			
8	`SW x4,8(x8)`	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).

Example (cycles 4, load execution)

		Decode	Issue	Complete	Commit
1	`LW x2,4(sp)`	1	2	5	
2	`LW x3,8(sp)`	1	3	6	
3	`ADD x1,x3,x2`	2			
4	`SW x1,0(sp)`	2			
5	`LW x5,0(x8)`	3	4	7	

Once insn 5's address is calculated (call it address X):

- Check SQ for completed, uncommitted stores to address X

"before I go to memory, are there any stores about to write to address X? If so, give me the value and I can avoid going to memory!"

- Write entry in LQ: insn 5 (address X) just loaded data from memory / from insn n in the SQ

Example (cycle 5)

		Decode	Issue	Complete	Commit
1	`LW x2,4(sp)`	1	2	5	
2	`LW x3,8(sp)`	1	3	6	
3	`ADD x1,x3,x2`	2			
4	`SW x1,0(sp)`	2			
5	`LW x5,0(x8)`	3	4	7	
6	`LW x6,4(x8)`	3	5	8	
7	`SUB x4,x5,x6`	4			
8	`SW x4,8(x8)`	4			

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

Speculatively execute #6 before the store (#4).

Again, check SQ and put entry in LQ

Example (cycle 6)

		Decode	Issue	Complete	Commit
1	`LW x2,4(sp)`	1	2	5	6
2	`LW x3,8(sp)`	1	3	6	
3	`ADD x1,x3,x2`	2	6	7	
4	`SW x1,0(sp)`	2			
5	`LW x5,0(x8)`	3	4	7	
6	`LW x6,4(x8)`	3	5	8	
7	`SUB x4,x5,x6`	4			
8	`SW x4,8(x8)`	4			

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

Insn 3 finally wakes up and is selected to issue

Example (cycle 7)

		Decode	Issue	Complete	Commit
1	`LW x2,4(sp)`	1	2	5	6
2	`LW x3,8(sp)`	1	3	6	7
3	`ADD x1,x3,x2`	2	6	7	
4	`SW x1,0(sp)`	2	7		
5	`LW x5,0(x8)`	3	4	7	
6	`LW x6,4(x8)`	3	5		
7	`SUB x4,x5,x6`	4			
8	`SW x4,8(x8)`	4			

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

Insn 4 wakes up and is selected to issue

Example (cycle 7, store execution)

		Decode	Issue	Complete	Commit
1	`LW x2,4(sp)`	1	2	5	6
2	`LW x3,8(sp)`	1	3	6	7
3	`ADD x1,x3,x2`	2	6	7	
4	`SW x1,0(sp)`	2	7		
5	`LW x5,0(x8)`	3	4	7	
6	`LW x6,4(x8)`	3	5		

Once insn 4's address is calculated (call it address Y):

- Check LQ for loads that might have speculatively executed

*"are there any younger loads that read from address Y? If so, **they should have gotten their values from insn 4** – squash them and give them my value!"*

- Write entry in SQ: insn 4 writes data D to address Y @ commit

Example (cycle 9)

		Decode	Issue	Complete	Commit
1	`LW x2,4(sp)`	1	2	5	6
2	`LW x3,8(sp)`	1	3	6	7
3	`ADD x1,x3,x2`	2	6	7	8
4	`SW x1,0(sp)`	2	7	8	9
5	`LW x5,0(x8)`	3	4	7	9
6	`LW x6,4(x8)`	3	5	8	
7	`SUB x4,x5,x6`	4	8	9	
8	`SW x4,8(x8)`	4	9		

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

Insn 8 wakes up and is selected to issue

Again, check LQ and put entry in SQ

Example (cycle 11)

		Decode	Issue	Complete	Commit
1	`LW x2,4(sp)`	1	2	5	6
2	`LW x3,8(sp)`	1	3	6	7
3	`ADD x1,x3,x2`	2	6	7	8
4	`SW x1,0(sp)`	2	7	8	9
5	`LW x5,0(x8)`	3	4	7	9
6	`LW x6,4(x8)`	3	5	8	10
7	`SUB x4,x5,x6`	4	8	9	10
8	`SW x4,8(x8)`	4	9	10	11

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

TaDa! Out of Order with memory instructions!

Aggressive Load Scheduling

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

  + When no conflict, increases performance

  – Conflict → may end up squashing a lot of instructions
    - High performance processors will learn which loads should issue early and which loads should wait.