Computer Architecture

Modern Out-of-Order Processors

Prof. Scott Beamer
sbeamer@ucsc.edu

CSE 220
https://canvas.ucsc.edu/courses/56561

Baskin
Engineering
UC SANTA CRUZ

Increasing Performance via ILP

- By Little's Law, to improve instruction throughput, if we can't improve latency, we will need to increase *parallelism*
- Types of parallelism
  - *Temporal* – multiple things in flight on same hardware – pipelining
  - *Spatial* – multiple hardware units – superscalar
- Challenge to executing more instructions at once is thus:
  - Identifying which instructions can execute in parallel
  - Providing instructions with the needed data
- ILP enhancement techniques in this course
  - *Pipelining* – including forwarding
  - *Caches* – fewer memory stalls means more instructions in flight
  - *Dynamic scheduling* – including scoreboarding (today) and Tomasulo
  - *Branch prediction* – speculate branches to uncover more instructions
  - *Memory prefetching* – further reduce memory stalls
  - *TLP* – use multiple threads/cores to execute more

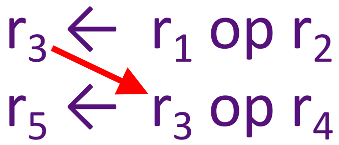
Types of Data Hazards

Consider executing a sequence of

$r_k \leftarrow r_i \ \text{op} \ r_j$

type of instructions

Data-dependence

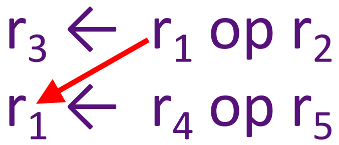


$r_3 \leftarrow r_1 \ \text{op} \ r_2$

$r_5 \leftarrow r_3 \ \text{op} \ r_4$

Read-after-Write (RAW) hazard

Anti-dependence

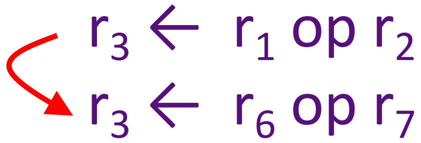


$r_3 \leftarrow r_1 \ \text{op} \ r_2$

$r_1 \leftarrow r_4 \ \text{op} \ r_5$

Write-after-Read (WAR) hazard

Output-dependence



$r_3 \leftarrow r_1 \ \text{op} \ r_2$

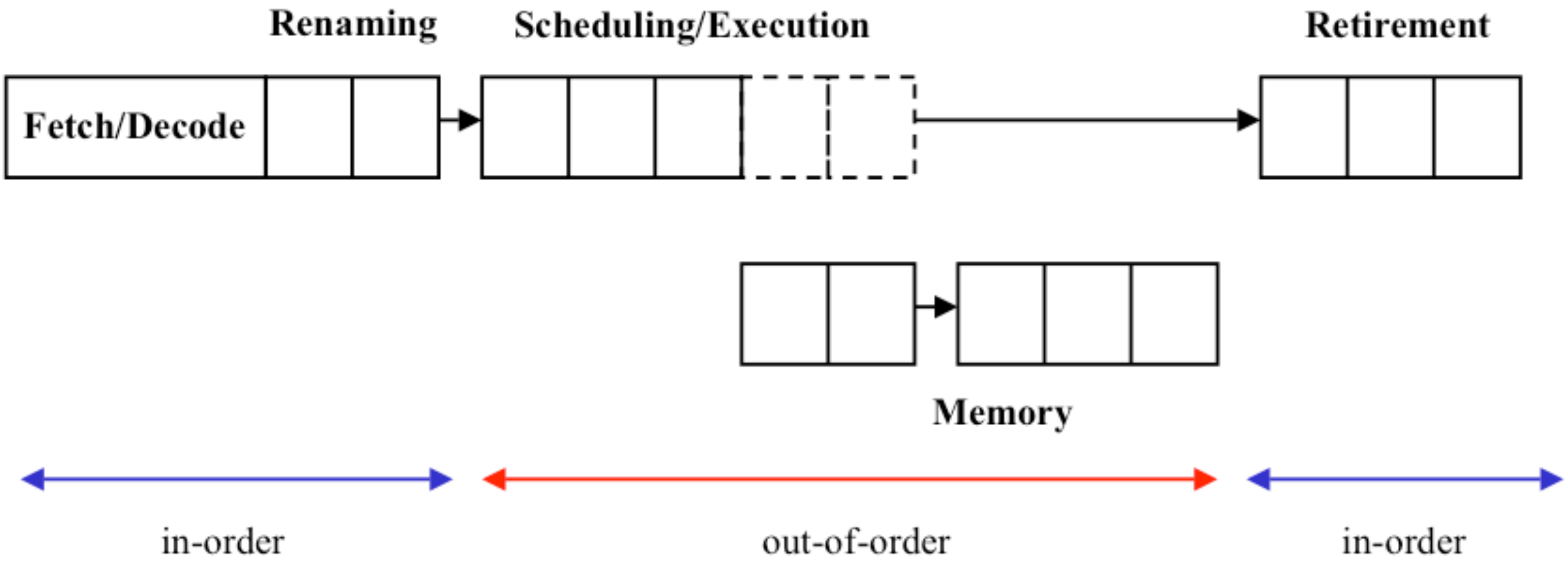
$r_3 \leftarrow r_6 \ \text{op} \ r_7$

Write-after-Write (WAW) hazard

Roadmap for Exploiting ILP

- Single-cycle CPU
  - Directly implements abstraction of in order & 1 instruction at a time
- Pipelined (e.g. 5-stage RISC)
  - Can overlap execution of multiple instructions, but everything in order
  - Forwarding helps with some RAW, in-order eliminates WAR/WAW
- Scoreboard
  - Fetch/Decode/Issue in-order, but allow out-of-order exec. & completion
  - Helpful for accommodating long latency functional units
  - Stalls for any hazard (RAW,WAR,WAW), but instr. can pass another
- Tomasulo
  - Uses register renaming to eliminate WAR & WAW
- Modern Out-of-Order (OoO)
  - Supports precise-exceptions
  - Simplifies/streamlines/centralizes OoO structures
  - Only limited by RAW hazards

Out-of-Order Processor Flow



Issue Window / Reservation Stations

- Hold instructions until executed
  - Instructions wait for operands and free functional unit

- Lifecyle of instruction in Issue Window
  - Inserted into window after rename (same time as into ROB)
  - *Wakeup* – operands become available (marked ready)
  - *Select* – chosen from ready instructions (and FU free) to execute
  - *Issue* – instruction was selected, sent to FU and removed

- Wakeup and select often combined into same pipeline stage

- Issue window is typically much smaller than ROB
  - ROB holds instructions until committed
  - Issue window holds instructions until start execution (issue)
  - Instructions remaining in issue window may not be contiguous

Modern OoO Example

```
ld   R1, 0(R2)
addi R2, R2, 8
sub  R3, R3, R1
bne  R2, R0, L
ld   R1, 0(R2)
```

Reorder Buffer (ROB)

	Op.	Logical Dest.	Physical Dest.	Physical Src. 1	Physical Src. 2	Executed
I_1						
I_2						
I_3						
I_4						
I_5						

Frontend RAT

R0	
R1	
R2	
R3	

Issue Window

Unit	Src.1	Src. 1 Ready	Src. 2	Src. 2 Ready

Free List:

Retirement RAT

R0	
R1	
R2	
R3	

Modern OoO Example

```
L:ld    R1,  0(R2)
  addi  R2,  R2,  8
  sub   R3,  R3,  R1
  bne   R2,  R0,  L
  ld    R1,  0(R2)
```

H →

Reorder Buffer (ROB)

	Op.	Logical Dest.	Physical Dest.	Physical Src. 1	Physical Src. 2	Executed
I1	ld	R1	P2	P3	i0	
I2						
I3						
I4						
I5						

Frontend RAT

R0	P0
R1	~~P6~~ P2
R2	P3
R3	P5

Issue Window

Unit	Src.1	Src. 1 Ready	Src. 2	Src. 2 Ready
MEM	P3	✓	i0	✓

Free List: ~~P2~~, P1, P7, P8, P9

Retirement RAT

R0	P0
R1	P6
R2	P3
R3	P5

Modern OoO Example

```
L:ld   R1, 0(R2)
  addi R2, R2, 8
  sub  R3, R3, R1
  bne  R2, R0, L
  ld   R1, 0(R2)
```

Reorder Buffer (ROB)

	Op.	Logical Dest.	Physical Dest.	Physical Src. 1	Physical Src. 2	Executed
T → I_1	ld	R1	P2	P3	i0	
I_2	addi	R2	P1	P3	i8	
H → I_3						
I_4						
I_5						

Frontend RAT

R0	P0
R1	~~P6~~ P2
R2	~~P3~~ P1
R3	P5

Issue Window

Unit	Src.1	Src. 1 Ready	Src. 2	Src. 2 Ready
ALU	P3	✓	i8	✓

Free List: ~~P2~~, ~~P1~~, P7, P8, P9

Retirement RAT

R0	P0
R1	P6
R2	P3
R3	P5

Modern OoO Example

```
L:ld   R1, 0(R2)
  addi R2, R2, 8
  sub  R3, R3, R1
  bne  R2, R0, L
  ld   R1, 0(R2)
```

Reorder Buffer (ROB)

	Op.	Logical Dest.	Physical Dest.	Physical Src. 1	Physical Src. 2	Executed
T → I1	ld	R1	P2	P3	i0	
I2	addi	R2	P1	P3	i8	
I3	sub	R3	P7	P5	P2	
H → I4						
I5						

Frontend RAT

R0	P0
R1	~~P6~~ P2
R2	~~P3~~ P1
R3	~~P5~~ P7

Issue Window

Unit	Src.1	Src. 1 Ready	Src. 2	Src. 2 Ready
ALU	P5	✓	P2	

Free List: ~~P2~~, ~~P1~~, ~~P7~~, P8, P9

Retirement RAT

R0	P0
R1	P6
R2	P3
R3	P5

Modern OoO Example

```
L:ld   R1, 0(R2)
  addi R2, R2, 8
  sub  R3, R3, R1
  bne  R2, R0, L
  ld   R1, 0(R2)
```

Reorder Buffer (ROB)

		Op.	Logical Dest.	Physical Dest.	Physical Src. 1	Physical Src. 2	Executed
T →	I1	ld	R1	P2	P3	i0	
	I2	addi	R2	P1	P3	i8	✓
	I3	sub	R3	P7	P5	P2	
	I4	bne	-	-	P1	P0	
H →	I5						

Frontend RAT

R0	P0
R1	~~P6~~ P2
R2	~~P3~~ P1
R3	~~P5~~ P7

Issue Window

Unit	Src.1	Src. 1 Ready	Src. 2	Src. 2 Ready
BRA	P1	✓	P0	✓
ALU	P5	✓	P2	

Free List: ~~P2~~, ~~P1~~, ~~P7~~, P8, P9

Retirement RAT

R0	P0
R1	P6
R2	P3
R3	P5

Modern OoO Example

```
L: ld   R1, 0(R2)
   addi R2, R2, 8
   sub  R3, R3, R1
   bne  R2, R0, L
   ld   R1, 0(R2)
```

Reorder Buffer (ROB)

	Op.	Logical Dest.	Physical Dest.	Physical Src. 1	Physical Src. 2	Executed
I_1	ld	R1	P2	P3	i0	✓
T → I_2	addi	R2	P1	P3	i8	✓
I_3	sub	R3	P7	P5	P2	
I_4	bne	–	–	P1	P0	
H → I_5	ld	R1	P8	P1	i0	

Frontend RAT

R0	P0
R1	~~P6~~ ~~P2~~ P8
R2	~~P3~~ P1
R3	~~P5~~ P7

Issue Window

Unit	Src.1	Src. 1 Ready	Src. 2	Src. 2 Ready
MEM	P1	✓	i0	✓
ALU	P5	✓	P2	✓

Free List: ~~P2~~, ~~P1~~, ~~P7~~, ~~P8~~, P9, P6

Retirement RAT

R0	P0
R1	~~P6~~ P2
R2	P3
R3	P5

Modern OoO Example

```
L:ld   R1, 0(R2)
  addi R2, R2, 8
  sub  R3, R3, R1
  bne  R2, R0, L
  ld   R1, 0(R2)
```

Reorder Buffer (ROB)

	Op.	Logical Dest.	Physical Dest.	Physical Src. 1	Physical Src. 2	Executed
H → I1						
T → I2	addi	R2	P1	P3	i 8	✓
I3	sub	R3	P7	P5	P2	
I4	bne	–	–	P1	P0	✓
I5	ld	R1	P8	P1	i 0	

Frontend RAT

R0	P0
R1	~~P6~~ ~~P2~~ P8
R2	~~P3~~ P1
R3	~~P5~~ P7

Issue Window

Unit	Src.1	Src. 1 Ready	Src. 2	Src. 2 Ready

Free List: ~~P2~~, ~~P1~~, ~~P7~~, ~~P8~~, P9, P6, P3

Retirement RAT

R0	P0
R1	~~P6~~ P2
R2	~~P3~~ P1
R3	P5

Modern OoO Example

```
L:ld   R1, 0(R2)
  addi R2, R2, 8
  sub  R3, R3, R1
  bne  R2, R0, L
  ld   R1, 0(R2)
```

Reorder Buffer (ROB)

	Op.	Logical Dest.	Physical Dest.	Physical Src. 1	Physical Src. 2	Executed
H → I1						
I2						
T → I3	sub	R3	P7	P5	P2	
I4	bne	–	–	P1	P0	✓
I5	ld	R1	P8	P1	i0	

Frontend RAT

R0	P0
R1	~~P6~~ ~~P2~~ P8
R2	~~P3~~ P1
R3	~~P5~~ P7

Issue Window

Unit	Src.1	Src. 1 Ready	Src. 2	Src. 2 Ready

Free List: ~~P2~~, ~~P1~~, ~~P7~~, ~~P8~~, P9, P6, P3

Retirement RAT

R0	P0
R1	~~P6~~ P2
R2	~~P3~~ P1
R3	P5

Modern OoO Example

```
L:ld   R1, 0(R2)
addi R2, R2, 8
sub  R3, R3, R1
bne  R2, R0, L
ld   R1, 0(R2)
```

Reorder Buffer (ROB)

	Op.	Logical Dest.	Physical Dest.	Physical Src. 1	Physical Src. 2	Executed
H → I_1						
I_2						
T → I_3	sub	R3	P7	P5	P2	✓
I_4	bne	–	–	P1	P0	✓
I_5	ld	R1	P8	P1	i0	

Frontend RAT

R0	P0
R1	~~P6~~ ~~P2~~ P8
R2	~~P3~~ P1
R3	~~P5~~ P7

Issue Window

Unit	Src.1	Src. 1 Ready	Src. 2	Src. 2 Ready

Free List: ~~P2~~, ~~P1~~, ~~P7~~, ~~P8~~, P9, P6, P3

Retirement RAT

R0	P0
R1	~~P6~~ P2
R2	~~P3~~ P1
R3	P5

Modern OoO Example

```
L:ld   R1, 0(R2)
  addi R2, R2, 8
  sub  R3, R3, R1
  bne  R2, R0, L
  ld   R1, 0(R2)
```

Reorder Buffer (ROB)

	Op.	Logical Dest.	Physical Dest.	Physical Src. 1	Physical Src. 2	Executed
H → I_1						
I_2						
I_3						
T → I_4	bne	-	-	P1	P0	✓
I_5	ld	R1	P8	P1	i0	

Frontend RAT

R0	P0
R1	~~P6~~ ~~P2~~ P8
R2	~~P3~~ P1
R3	~~P5~~ P7

Issue Window

Unit	Src.1	Src. 1 Ready	Src. 2	Src. 2 Ready

Free List: ~~P2~~, ~~P1~~, ~~P7~~, ~~P8~~, P9, P6, P3, P5

Retirement RAT

R0	P0
R1	~~P6~~ P2
R2	~~P3~~ P1
R3	~~P5~~ P7

Exception Handling

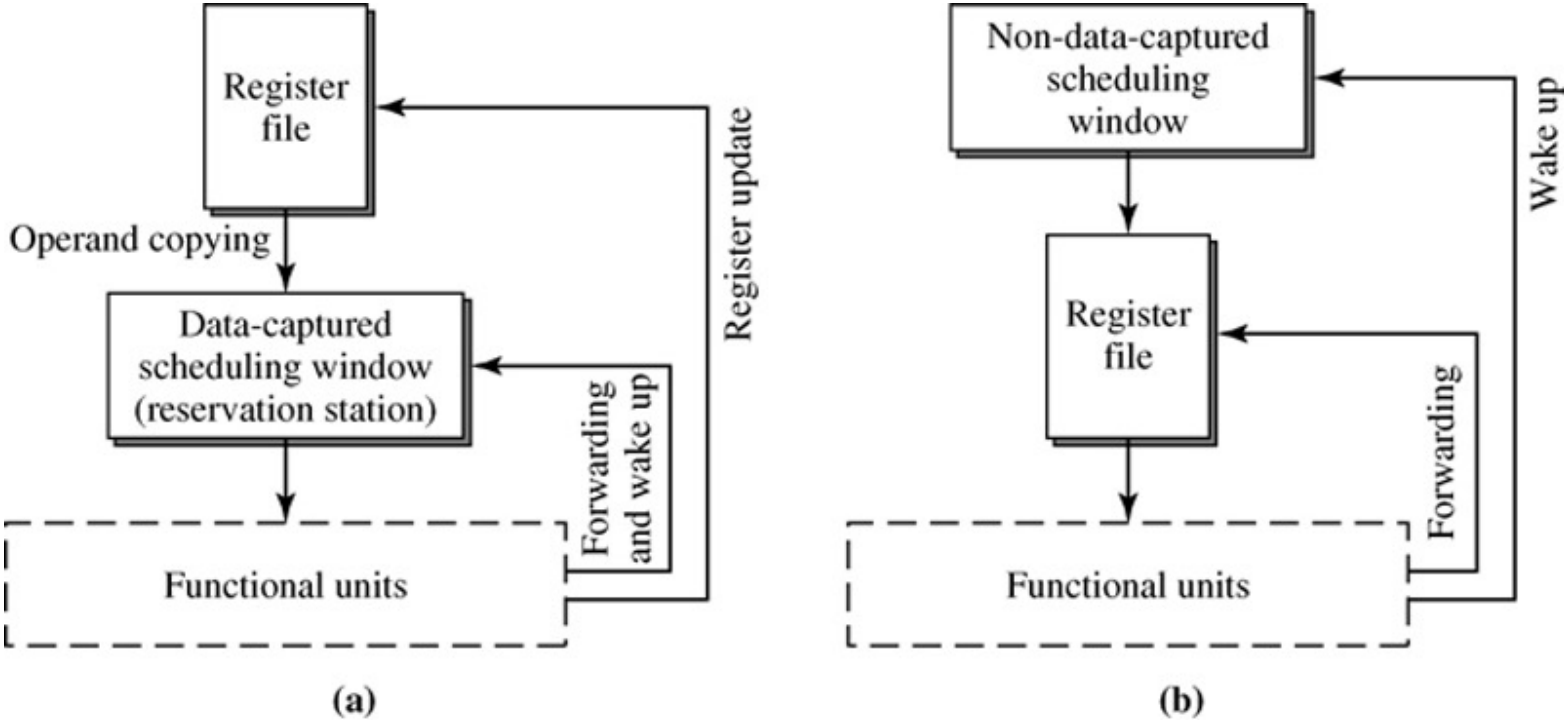
- Issue Logic executes out-of-order, ROB brings original order back
- *Key: no store visible outside the CPU until retirement/commit*
  - all previous instructions committed
    - architectural state up to date
  - this and subsequent instructions not committed
    - architectural state not compromised
  - false exceptions from mispredicted path never emerge
- Once excepting instruction makes it to commit stage:
  - flush all uncommitted instructions
  - save precise architectural state
  - handle exception
  - restore architectural state and initiate fetch

Peer Instruction Question

To count how many instructions in flight write to the same architectural register, you should check:

- A) Count values that match in Frontend RAT
- B) Count values that match in Retirement RAT
- C) Count # entries in ROB with matching logical destination
- D) Sum of all of above

When to Read the Register File?



Data in ROB

Unified Physical Register File

Timeline Samples

Modern OoO, no FWD

```
  a=b+c  IF RN WS R0 R1 EX WB
  d=a+3     IF RN -- -- -- WS R0 R1 EX WB
```

Modern OoO with FWD

```
  a=b+c  IF RN WS R0 R1 EX WB
  d=a+3     IF RN WS R0 R1 EX WB
```

Modern OoO with FWD

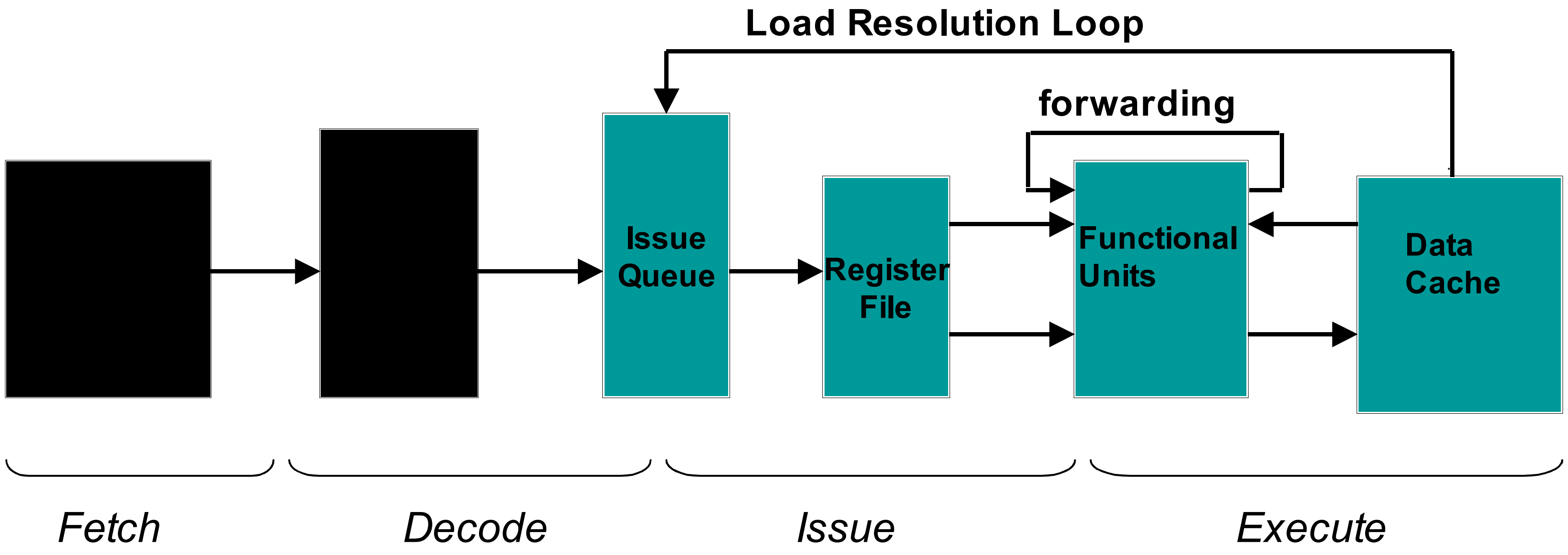
```
  a=a*b  IF RN WS R0 R1 E0 E1 WB
  d=a+3     IF RN -- WS R0 R1 EX WB
```

Modern OoO with FWD (no load hit speculation)

```
  a=ld(f)IF RN WS R0 R1 M0 M1 WB
  d=a+3     IF RN -- -- -- -- WS R0 R1 EX WB
```

Load-Hit Speculation

- We can not wait to wakeup dependent instructions until load finish
- Must have feature for in-order and out-order processors
  - Becoming more important as the pipeline depth increases



Timeline Samples

Modern OoO, no FWD

```
  a=b+c  IF RN WS R0 R1 EX WB
  d=a+3     IF RN RN RN RN WS R0 R1 EX WB
```

Modern OoO with FWD

```
  a=b+c  IF RN WS R0 R1 EX WB
  d=a+3     IF RN WS R0 R1 EX WB
```

Modern OoO with FWD

```
  a=a*b  IF RN WS R0 R1 E0 E1 WB
  d=a+3     IF RN RN WS R0 R1 EX WB
```

Modern OoO with FWD (no load hit speculation)

```
  a=ld(f)IF RN WS R0 R1 M0 M1 WB
  d=a+3     IF RN RN RN RN RN WS R0 R1 EX WB
```

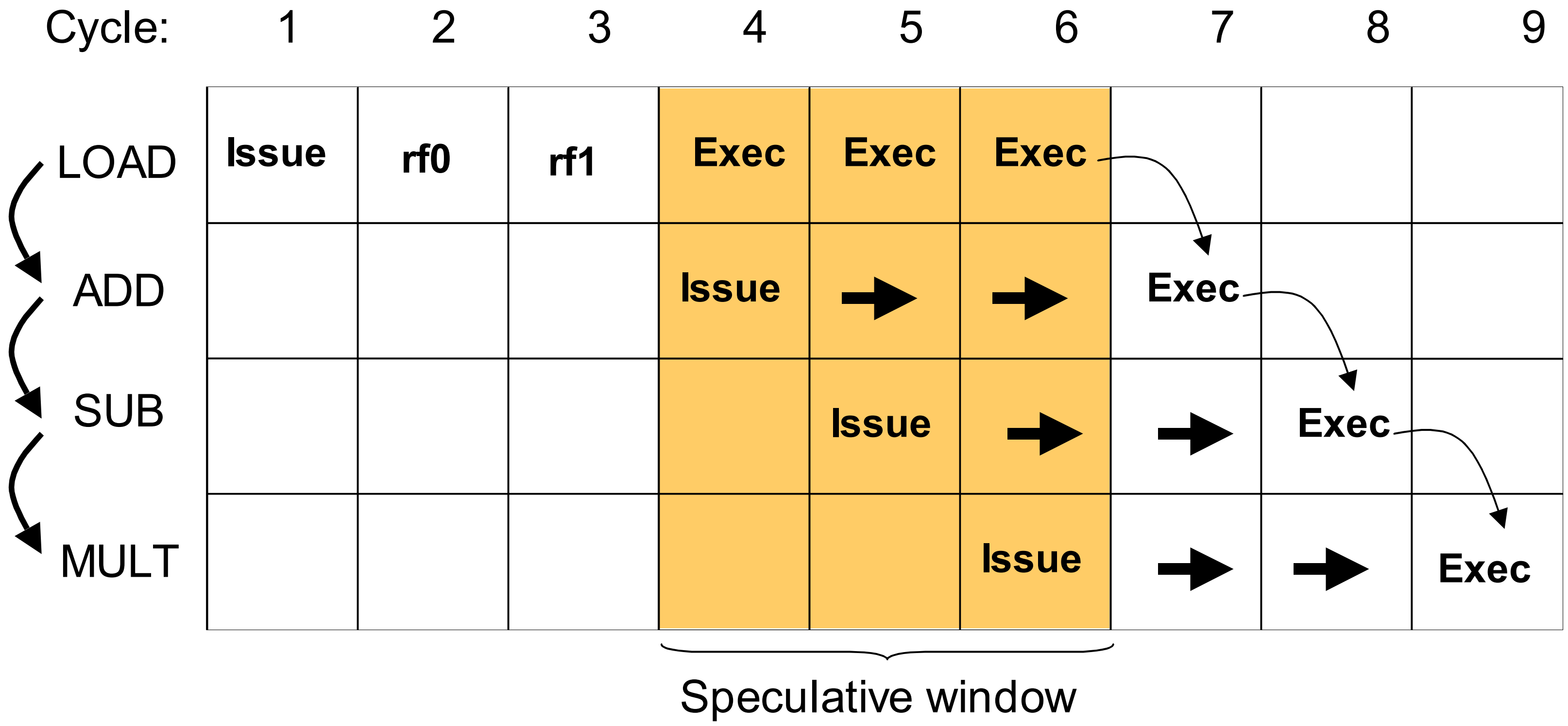
Modern OoO with FWD (load hit speculation with cache hit)

```
  a=ld(f)IF RN WS R0 R1 M0 M1 WB
  d=a+3     IF RN RN WS R0 R1 EX WB
```

Modern OoO with FWD (load hit speculation with cache miss)

```
  a=ld(f)IF RN WS R0 R1 M0 M1 M2 M3 WB
  d=a+3     IF RN RN WS R0 R1 EX WB
  d=a+3           -- -- -- -- -- -- WS R0 R1 EX WB
```

Load-Hit Speculation: Speculative Window

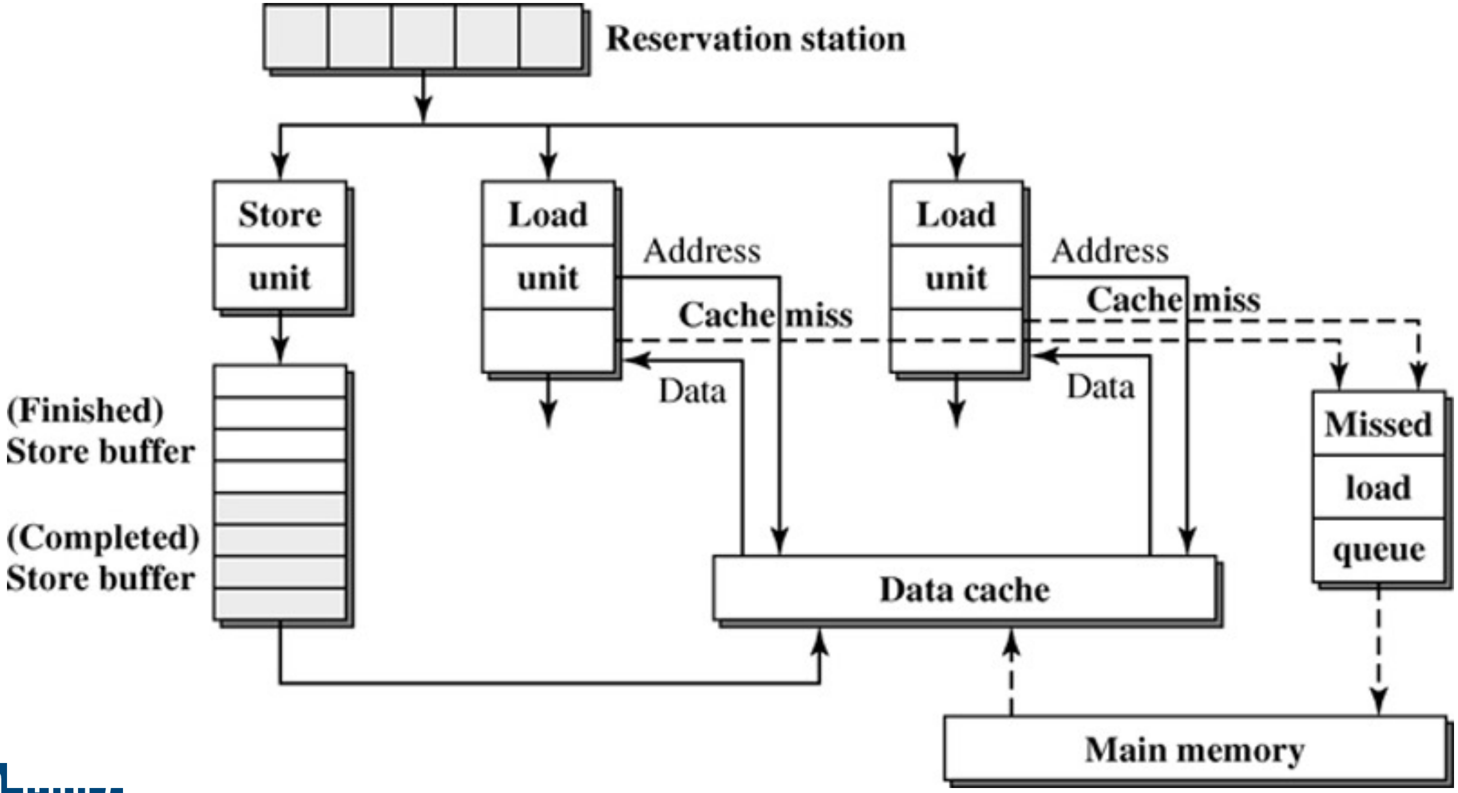


Cycle:	1	2	3	4	5	6	7	8	9
LOAD	Issue	rf0	rf1	Exec	Exec	Exec			
ADD				Issue	→	→	Exec		
SUB					Issue	→	→	Exec	
MULT						Issue	→	→	Exec

Speculative window

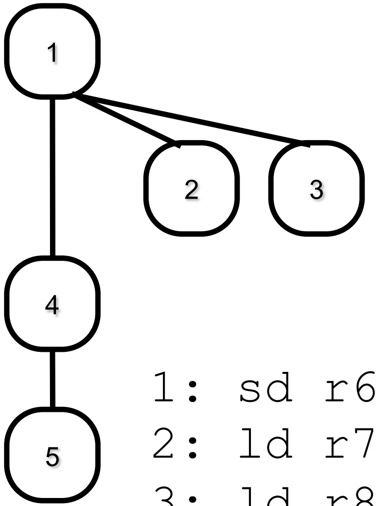
Memory Speculation, Why should I care?

- Can you exec. memory instructions (loads & stores) out-of-order?
- Between 10-30% slowdown with no memory speculation
- All high performance OoO cores implement it



No Memory Speculation Design

- Naïve - serialize all stores and loads
  - Exploits less memory-level parallelism
- Want to start loads sooner when possible
- When is it safe to load?
  - When confident no prior (older) store will write to same address (*alias*)
  - Thus, need to know not only load address, but also all prior store addresses
  - This is conservative OoO



```
1: sd r6, 0(r1)
2: ld r7, 0(r2)
3: ld r8, 0(r3)
4: sd r9, 0(r4)
5: sd r10, 0(r5)
```

Load/Store Ordering

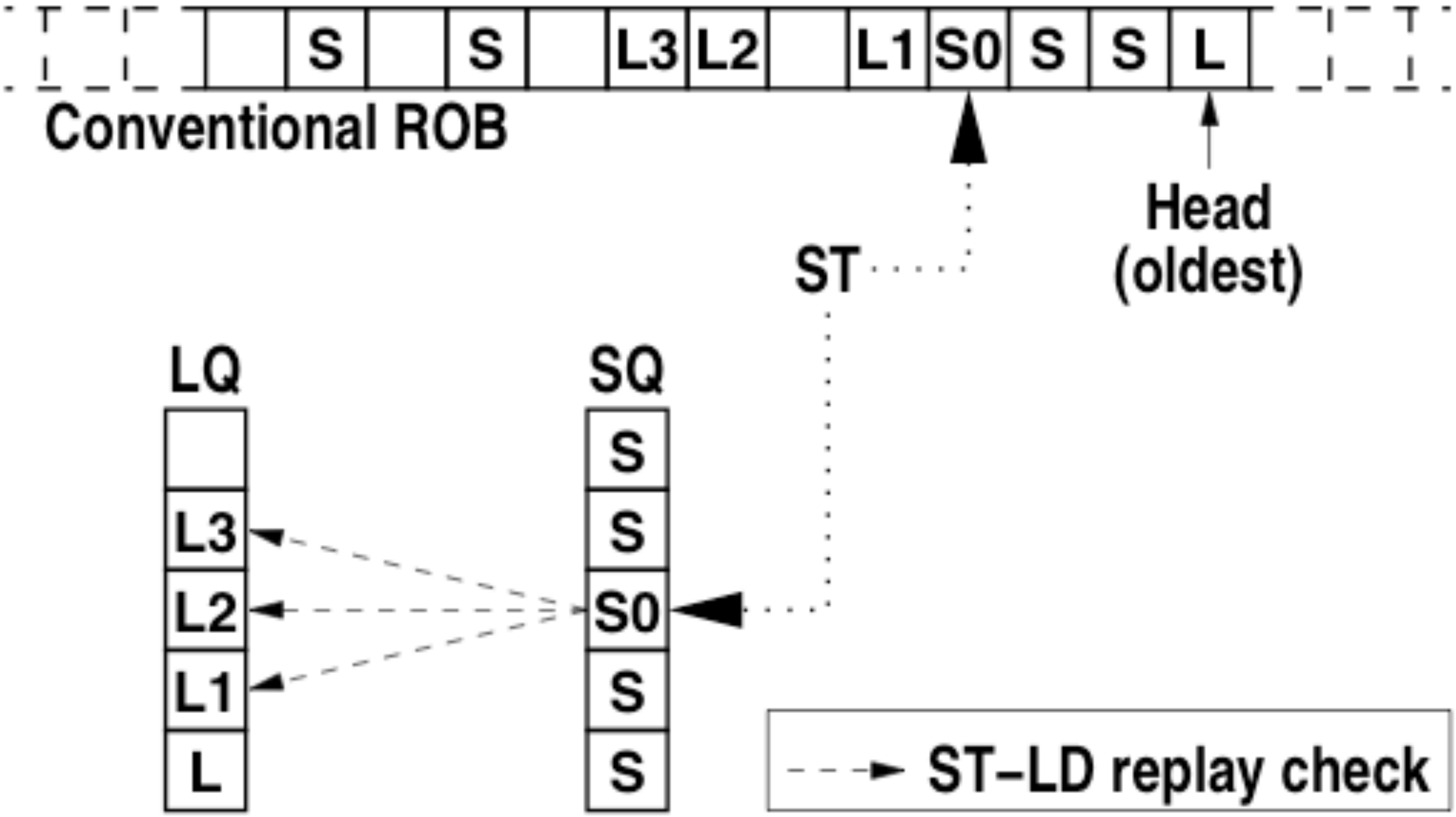
- Loads may execute out of order
  - starting loads early diminishes impact of miss penalty
- Stores always perform at commit
  - WAW – taken care of by in-order commit
- Must respect store-load ordering (ST-LD Replay)
  - RAW – must not execute (load) before store completes
    - could forward value from store buffer (not supported here)
  - WAR – automatically enforced by in-order stores at commit
- RAW enforcement: do not issue load if
  - earlier stores to overlapping address
  - *earlier stores to unresolved address*

Memory Speculation

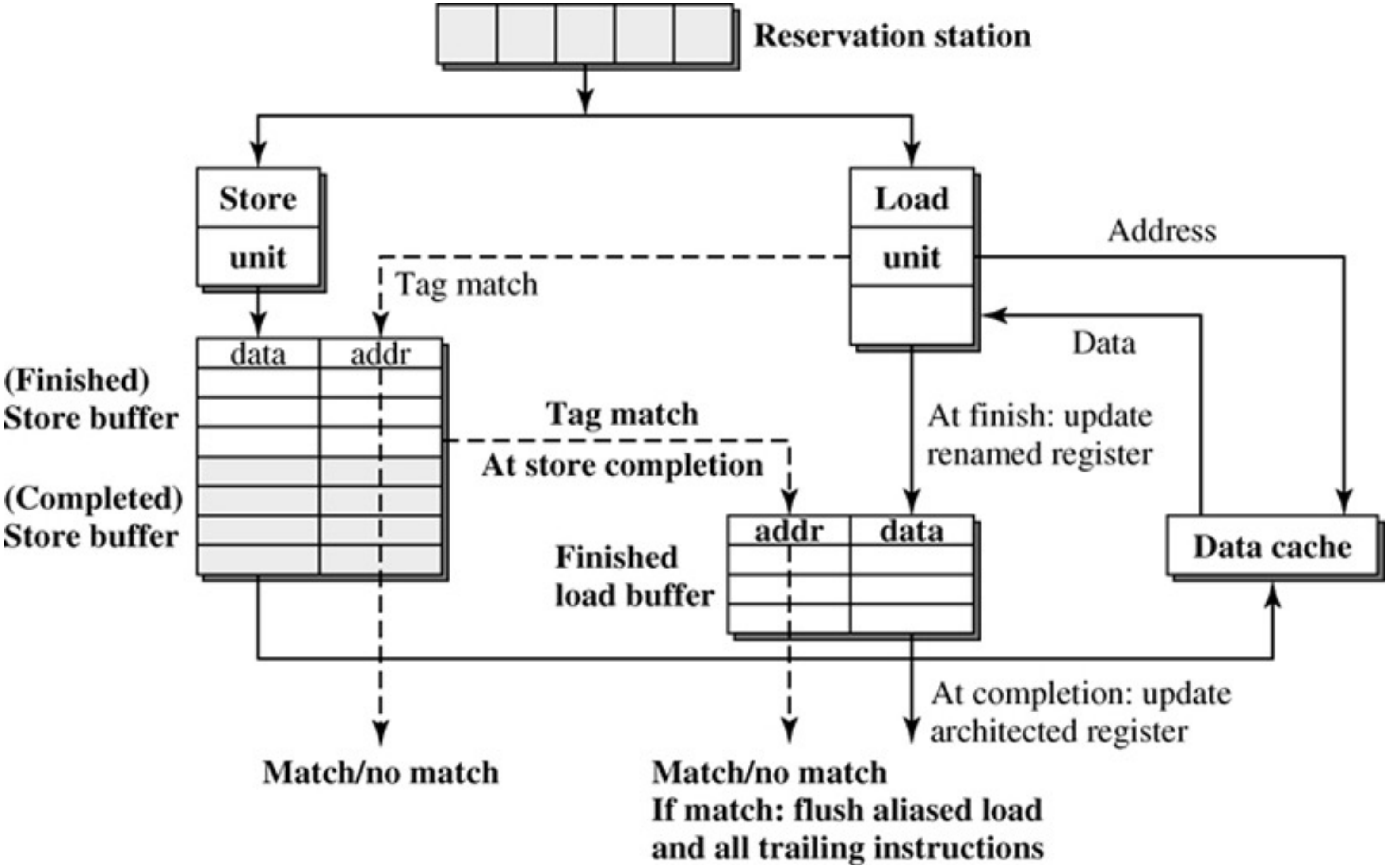
- If a prior store address is unknown, can *speculate* that the load will not have the same address (alias)
  - If speculation is right, got more things in flight (yay!)
  - If speculation is wrong, will need to recover

- To speculate, will need the following:
  - Load Queue (LQ)
    - Keeps all the loads addresses
  - Store Queue (SQ)
    - Keeps all the stores addresses
  - Store Data Queue (SDQ)
    - Queue associated with the store queue. Keeps the stores data

ST-LD Replay Check



MPD Diagram for LSQ



Store Completion Buffer (SCB)

- Holds stores (data and address) after retirement
  - Once store is in SCB, is retired
- SCB is essentially a store buffer (as covered in cache lectures)
- Holding store in buffer helps…
  - Hide variable latency of cache, so don’t want to slow retire
    - Could be a cache miss (on a system that is write-allocate & write-back)
  - Coalesce multiple stores to same address range
- Visibility of store
  - Pre-retirement – nobody outside processor can see store
    - Instructions within processor can read data via forwarding from SDQ
  - Post-retirement – “written” to memory
    - Requests for that data will check cache and SCB

Memory Speculation Summary

- Allow loads to start early by *speculating* will not alias with prior stores to unknown addresses

- Load remains in LQ until committed
- When store commits, checks for alias in LQ
  - If no match, safe
  - If addresses match, perform recovery
    - Load should have read data from store since same address
    - If load was before store in program order, would already be out of LQ

- Remember: *commit stage is in order*
  - If load was earlier in program order (WAR), will commit first, and will be gone from LQ when store commits
  - If store was earlier in program order (RAW), the process above double checks load got right value (and flushes if wrong)
- Read *Modern Processor Design* 5.3.2 – 5.3.3 for more

Acknowledgements

- These slides contain course material initially created by Jose Renau for CMPE 202