EE 457 Unit 9b

Tomasulo Part 2:

In-Order Completion

Speculation

Credits

- Some of the material in this presentation is taken from:
  - Computer Architecture: A Quantitative Approach
    - John Hennessy & David Patterson
- Some of the material in this presentation is derived from course notes and slides from
  - Prof. Michel Dubois (USC)
  - Prof. Murali Annavaram (USC)
  - Prof. David Patterson (UC Berkeley)

Tomasulo w/ Speculative Execution

Tomasulo 1

- In-order Issue
- Out-of-Order Execution
- *Out-of-order* Completion
  - Completion = Commit = Update state = Write to Reg./Mem.

- No speculative execution beyond branches (stall dispatch until branch is resolved)
- No precise exceptions

Tomasulo 2

- In-order Issue
- Out-of-Order Execution
- _______________ Completion
  - Plus, we now allow "Speculative" Execution

- Execute out of order but don't write reg/memory immediately but "________" (________________ store) results and commit in-order.
- Can speculate branch outcomes and dispatch down a pathway before they execute, flushing instruction results if we are wrong
- Support precise exceptions!

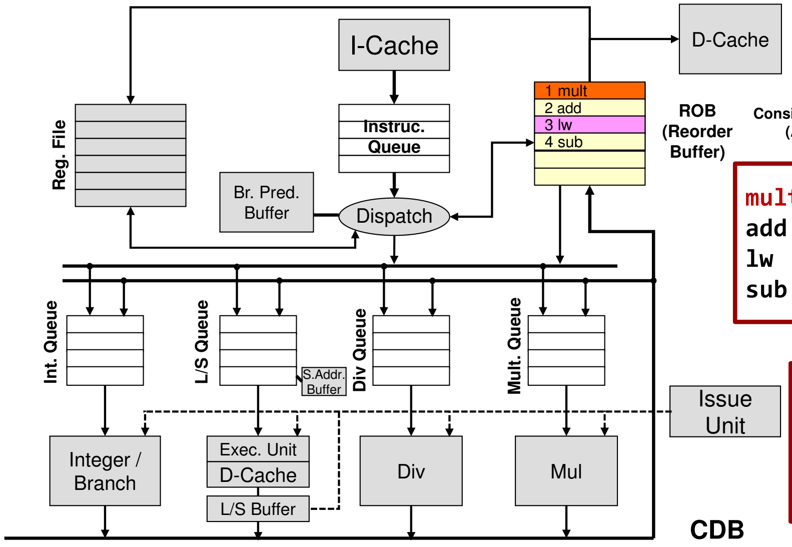
Changes to Tomasulo Part 1

- Removed structures:
  - No more **TAG FIFO:** Use ROB location (write pointer) as TAG of the instruction
  - No more **RST (Register Status Table): I**nstead do an associative search of the ROB
- D-Cache shown in one place (used by LW and SW in same place)

- New Structures:
  - ROB (_______________): Enables in-order completion and __________ after misspeculated branch
  - **BPB (Branch Prediction Buffer)**: Enables speculating (issuing instructions) past branches
  - **SAB (Store Address Buffer)**: Helps with memory disambiguation
- D-Cache shown in two places (LW and SW use at different places/times)

OoO Execution w/ ROB

- ROB allows for OoO execution but in-order completion



Assume: SW always hits in cache

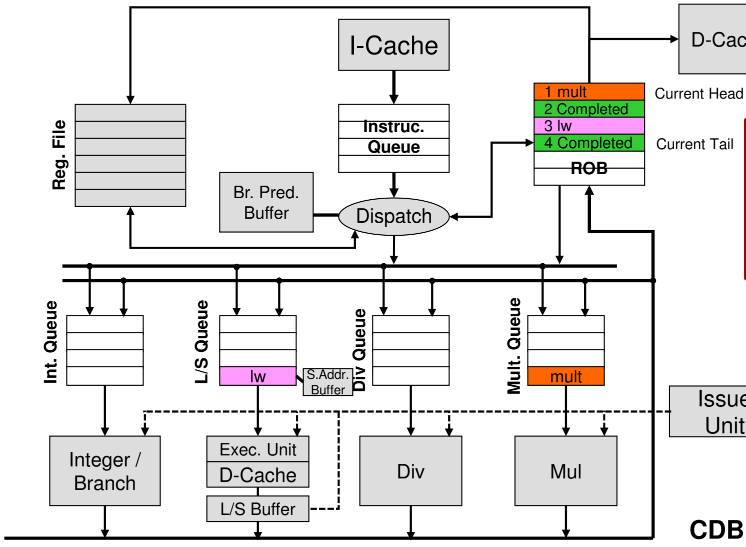
Consider this sequence: (Assume mult takes several cycles)

```
mult $5,$6,$7
add  $2,$3,$4
lw   $8,0($5)
sub  $9,$0,$2
```

Simplification for EE457: Cache miss can occur for LW only but SW always hits (without this simplification we need to cover store buffer design and related issues)

OoO Execution w/ ROB

- ROB allows for OoO execution but in-order completion



```
mult $5,$6,$7
add  $2,$3,$4
lw   $8,0($5)
sub  $9,$0,$2
```

ROB entry is allocated on dispatch.

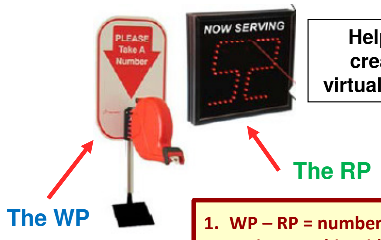
When an instruction executes, its result is stored in ROB then committed to register file when it reaches the head of the ROB (in-order completion)

Handling Data Dependencies and Enforcing In-Order Completion

REORDER BUFFER (ROB)

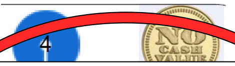
Take a Number vs. Take a Token

- ROB (Re-order Buffer) forms a virtual ____________ to maintain order (so we can complete in order)
- Take a number (WP) on ___________, and commit when you reach the ______________ (RP) and are ready
- ROB Tag = Paper token taken by the customer
  - Recall that we wrap back to 0 after the maximum tag number



Helps to create a virtual queue.

1. WP – RP = number of items in the FIFO (depth)
2. It is a circular FIFO/buffer



In State Bank of India, the cashier issues brass token to customers trying to draw money as an ID (and not at all to put them in any virtual queue / ordering). Token numbers are in random order.

The cashier verifies the signature in the record rooms, returns with money, calls the token number and issues the money.

Tokens are reclaimed & reused.

Re-Order Buffer (ROB) Structure

- ROB is a **FIFO** + ___________ Access
  - In a modern system: 128-256 locations
- WP = Write pointer
  - Used by Dispatch Unit
  - Each instruction issues in order and "takes a number" (its "_____")
- Instructions can **write results to its ROB entry** (out of order) whenever they execute and put their result on the CDB
- RP = Read pointer =
  - Used for committing (allow write-back for) the most senior / oldest instruction when it has completed without generating an exception

		Valid	Comp	Rd	RegWr	Result	Others
	0	0	0	0	1		
	1	0	0	$2	1		
	2	0	0	0	0		
Top (rp) →	3	1	1	$1	1		
	4	1	0	$2	1		
	5	1	0	$15	1		
	6	1	1	$2	1		
	7	1	1	$6	1		
	8	1	0	$2	0		
Bottom (wp) →	9	1	0	$7	0		
	10	0	0	$13	1		
	11	0	0	0	1		
	12	0	0	$4	0		
	13	0	0	$2	1		
	14	0	0	0	1		
	15	0	0	0	0		

Note: Valid is not needed (uses items from RP to WP)

Others: MemWrite (SW), MemAddr

Re-Order Buffer (ROB) Structure

- We will not use the RST (Register Status Table)
  - Though this may vary depending on implementation
- On **instruction dispatch**: the ROB is searched for its source register (Rs and/or Rt) producers and can find its source operands from one of three sources:
- **Unproduced** (e.g. add $8, **$2, $2**)
  - Situation: producer still waiting to ___________
  - Action: Take __________ of producer (**ROB8**)
- **Produced** (e.g. add $8, **$6, $6**)
  - Situation: Producer executed and is waiting to _____________
  - Action: Take ______ from ROB (**data from ROB7**)
- **Unfound** (e.g. add $8, **$3, $3**)
  - Situation: Latest value is in ___________
  - Action: Take value from RegFile
- Since multiple entries in the ROB may match Rs/Rt a priority resolver is necessary (e.g. $2)

		Valid	Comp	Rd	RegWr	Result	Others
	0	0	0	0	1		
	1	0	0	$2	1		
	2	0	0	0	0		
Top (rp) →	3	1	1	$1	1		
	4	1	0	$2	1		
	5	1	0	$15	1		
	6	1	1	$2	1		
	7	1	1	$6	1		
	8	1	0	$2	0		
Bottom (wp) →	9	1	0	$7	0		
	10	0	0	$13	1		
	11	0	0	0	1		
	12	0	0	$4	0		
	13	0	0	$2	1		
	14	0	0	0	1		
	15	0	0	0	0		

Note: Valid is not needed (uses items from RP to WP)

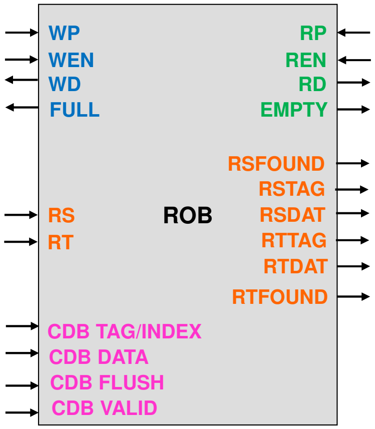
Others: MemWrite (SW), MemAddr

Dispatch and the ROB

- No more token FIFO (for tagging instructions) as in OoO execution and completion
  - _________ is your ___ and is allocated for an instruction on issue/dispatch
  - When instruction finishes executing its result is buffered in the ROB entry until it can be committed safely
- It does not use the RST (Register Status Table) as before (because of difficult with implementing speculative execution)
  - When an instruction is dispatched, the ROB is searched for its source register (Rs and/or Rt) producers
    - **Unproduced**: If an entry in the ROB is producing Rs/Rt but has **NOT YET EXECUTED** the ROB tag/slot of the producer is taken with the dependent instruction
    - **Produced**: If an entry in the ROB is producing Rs/Rt and the result is **PRODUCED BUT WAITING TO BE COMMITTED**, that value is taken with the dependent instruction
    - **Unfound**: If no entry in the ROB is producing Rs/Rt, **DATA IN THE REGISTER FILE IS THE LATEST** value and is taken with the dependent instruction
  - Since multiple entries in the ROB may match Rs/Rt a priority resolver is necessary

Not Just a FIFO: ROB Interfaces

- ROB has many interfaces
  - RP, WP work like a FIFO (sequential access)
  - RS,RT source register/tag lookup (associative search)
  - CDB write execution results (index / random access)



ROB DEPTH AND PRIORITY RESOLUTION

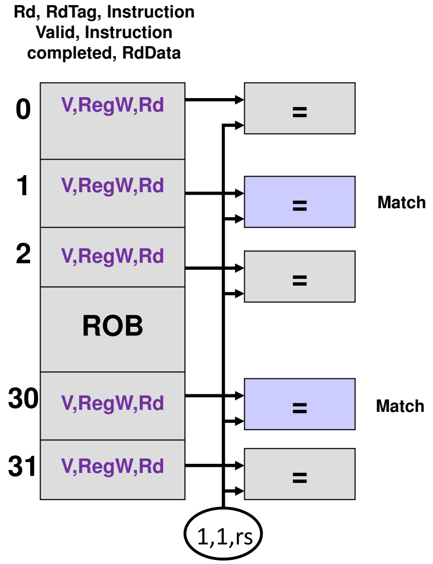
Motivation for finding ROB Depth

- How do we determine the correct ROB entry to help when trying to obtain our source registers
  - e.g. add $8, **$2, $2**
- We need to understand ROB depth calculation and priority resolution
- In the diagram how many instructions are waiting in the ROB?
  - Answer: _____________
- Can we just use the **LARGEST** valid index that matches the desired register? _______

		Valid	Comp	Rd	RegWr	Result	Others
	0	0	0	0	1		
	1	0	0	$2	1		
	2	0	0	0	0		
Top (rp) →	3	1	1	$1	1		
	4	1	0	$2	1		
	5	1	0	$15	1		
	6	1	1	$2	1		
	7	1	1	$6	1		
	8	1	0	$2	0		
Bottom (wp) →	9	1	0	$7	0		
	10	0	0	$13	1		
	11	0	0	$0	1		
	12	0	0	$4	0		
	13	0	0	$5	1		
	14	0	0	$9	1		
	15	0	0	$0	0		

ROB Matches

- Can we just use the **LARGEST** valid index that matches the desired register?
  - In the example to the right should we say to use entry 30's information?
- Not necessarily
  - Need to know where the ___ and ____ are
  - What if RP=30 and WP = 2?
  - Let's explore more



ROB Depth/Distance

- Case 1
  - Your number is 55 and mine is 65
  - I am ______ numbers (**after / before**) you.
- Case 2
  - Your number is 55 and mine is 45
  - I am ______ numbers (**after / before**) you.

Now serving:

52

Computing Distance

- To find how many people are waiting subtract the **Now Serving** number from the last number pulled
- Example
  - Last number pulled = 92
  - **Now Serving** = 52
  - # Waiting = _________________
- But suppose the last number pulled is 32
  - Last number pulled = 32
  - **Now Serving** = 52
  - # Waiting = _________________

Now serving:

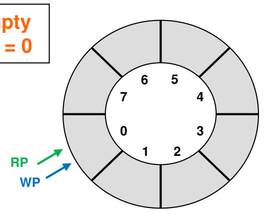
52

DEPTH = (WP-RP) __________

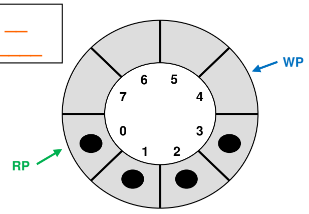
Computing Distance

- **Depth = (WP – RP) mod 8**

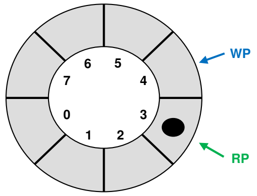
FIFO Initially Empty
D = WP-RP = 0-0 = 0



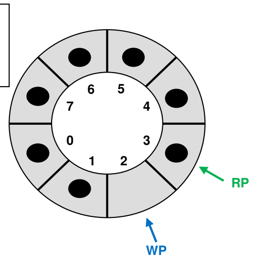
FIFO Depth = __
D = WP-RP = _____



FIFO Depth = ___
D = WP-RP = ______



FIFO Depth = ___
D = WP-RP
= ____________



ROB Dispatch for Rs

- $2 is needed by dispatch, which ROB entry should be selected as the producer?
- We want the ____________ producer
  - Prof. Puvvada would say "the most ______ (youngest) of our _______ (those before us)

Scenario 0

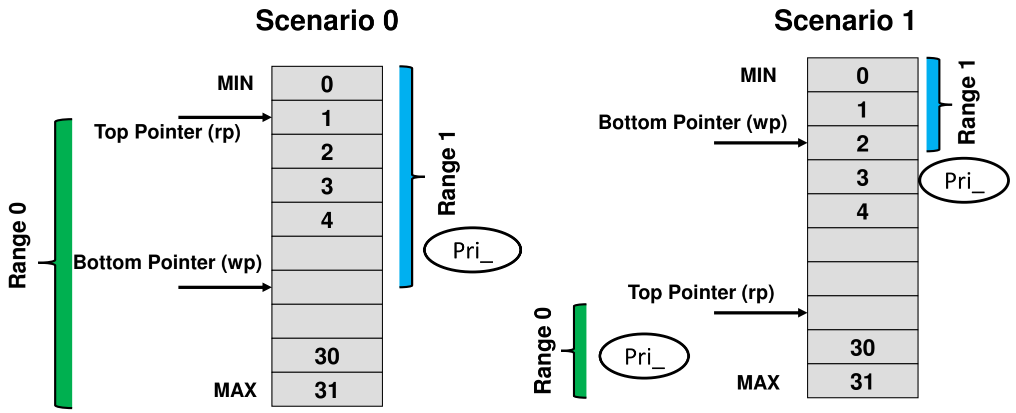
		Valid	Rd	RegWrite
	0	0	0	1
	1	0	$2	1
	2	0	0	0
Top (rp) →	3	1	$1	1
	4	1	$2	1
	5	1	$15	1
	6	1	$2	1
	7	1	$12	1
	8	1	$2	0
	9	1	$7	0
Bottom (wp) →	10	0	$13	1
	11	0	0	1
	12	0	$4	0
	13	0	$2	1
	14	0	0	1

Scenario 1

		Valid	Rd	RegWrite
	0	1	0	1
	1	1	$2	1
	2	1	$10	1
Bottom (wp) →	3	0	$1	0
	4	0	$21	1
	5	0	$12	1
	6	0	$2	0
	7	0	$15	1
	8	0	$22	1
Top (rp) →	9	1	$7	1
	10	1	$13	0
	11	1	$2	1
	12	1	$1	1
	13	1	$2	0
	14	1	$3	1

Dealing with Wrapping

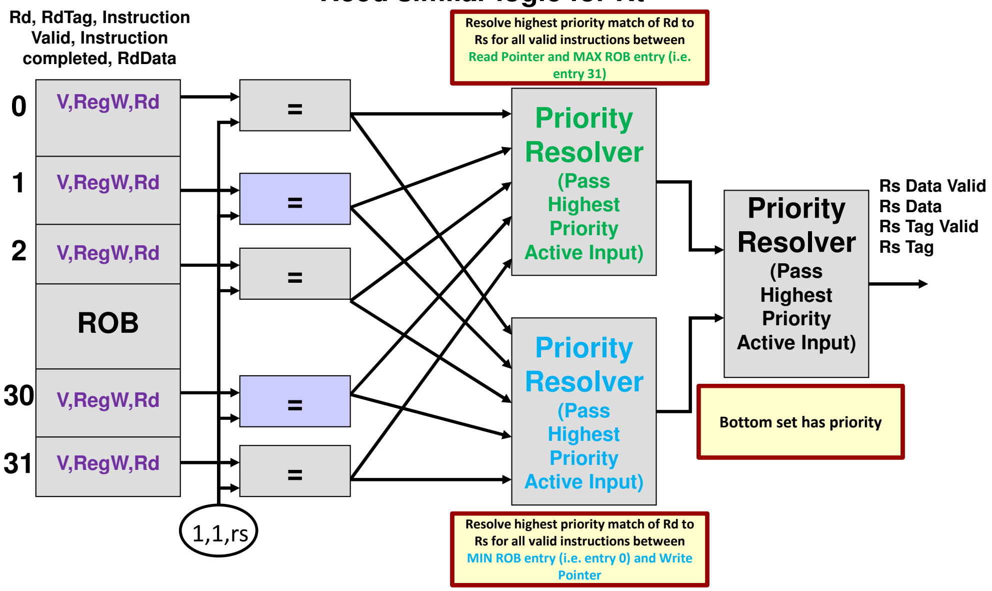
- Consider ranges: RP to MAX and MIN to WP



In each scenario, which set should be given higher priority of selection to forward the value of a particular register?

ROB Dispatch for Rs

Need similar logic for Rt


Rd, RdTag, Instruction Valid, Instruction completed, RdData
0
1
2
30
31
V,RegW,Rd
V,RegW,Rd
V,RegW,Rd
ROB
V,RegW,Rd
V,RegW,Rd
=
=
=
=
=
1,1,rs
Priority Resolver (Pass Highest Priority Active Input)
Priority Resolver (Pass Highest Priority Active Input)
Priority Resolver (Pass Highest Priority Active Input)
Rs Data Valid
Rs Data
Rs Tag Valid
Rs Tag

Resolve highest priority match of Rd to Rs for all valid instructions between Read Pointer and MAX ROB entry (i.e. entry 31)

Bottom set has priority

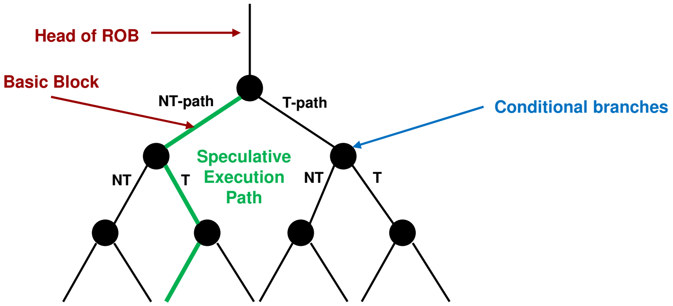
Resolve highest priority match of Rd to Rs for all valid instructions between MIN ROB entry (i.e. entry 0) and Write Pointer

Avoiding stalls for control hazards

BRANCH PREDICTION AND SPECULATIVE EXECUTION

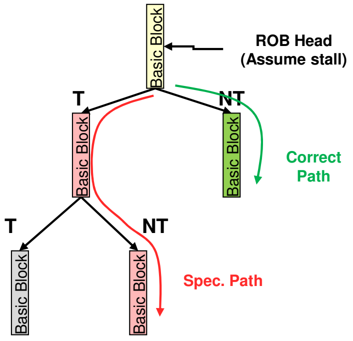
Branch Prediction + Speculation

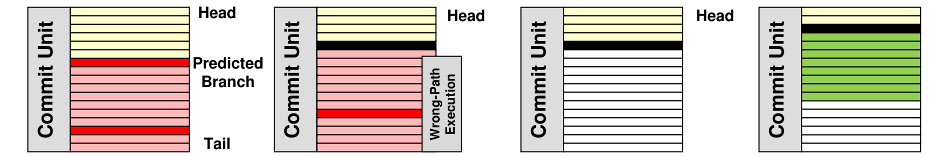
- To keep the backend fed with enough work we need to __________ a branch's outcome and perform "_________________" execution beyond the predicted (unresolved) branch
  - Roll back mechanism (flush) in case of misprediction



Speculation Example

- Predict branches and execute most likely path
  - Flush ROB entries and issue queues after the mispredicted branch
  - Need good prediction capabilities to make this useful





Time 1: ROB
Red Entries = Predicted Branches

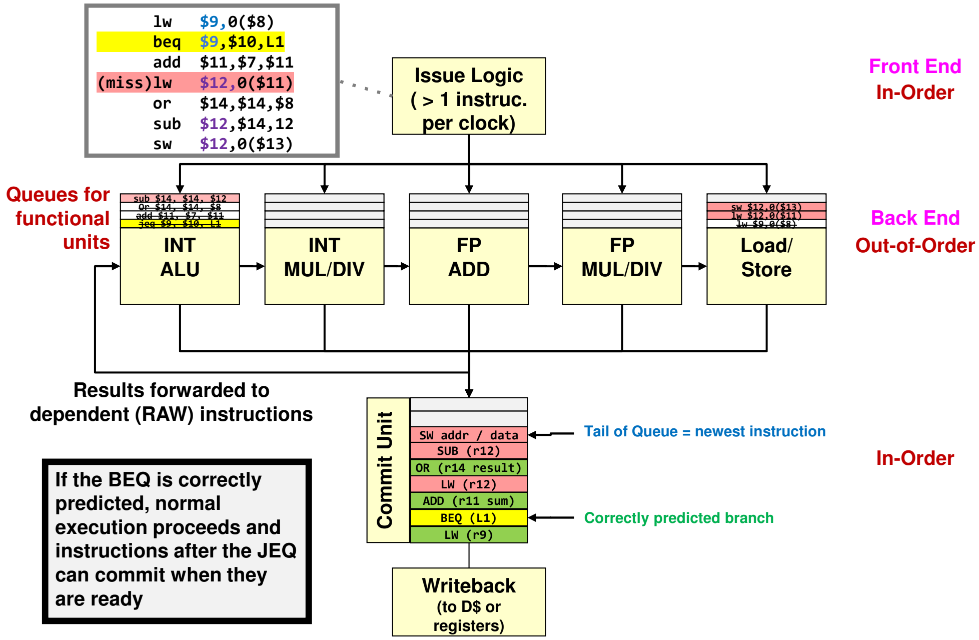
Time 2a: ROB
Black Entry = Mispredicted branch

Time 2b:
Flush ROB/Pipeline of instructions behind it

Time 3: ROB
Pipeline begins to fill w/ correct path

Case 1: Correct Prediction

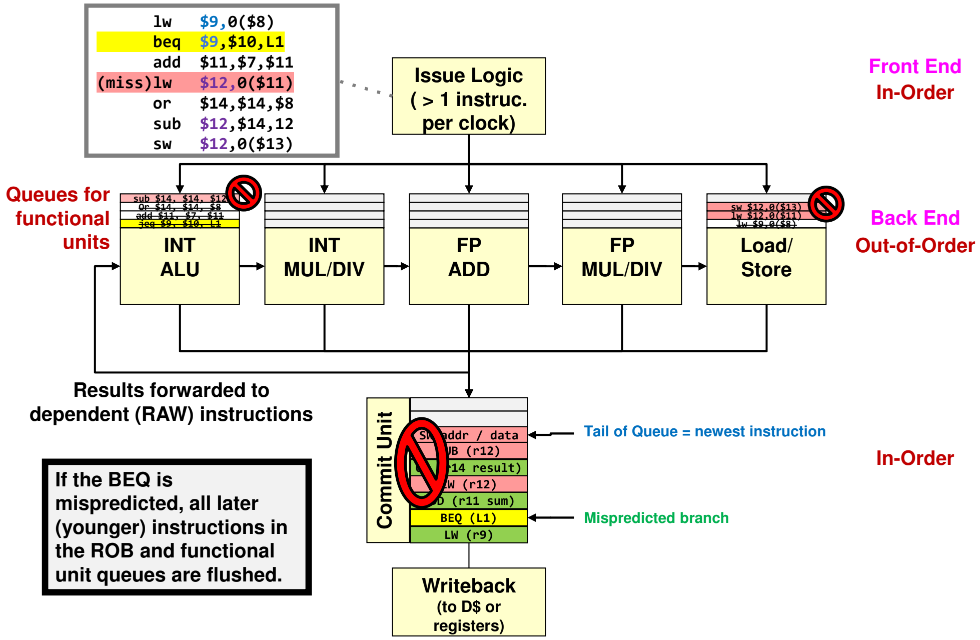
```
        lw   $9,0($8)
        beq  $9,$10,L1
        add  $11,$7,$11
(miss)lw     $12,0($11)
        or   $14,$14,$8
        sub  $12,$14,12
        sw   $12,0($13)
```



If the BEQ is correctly predicted, normal execution proceeds and instructions after the JEQ can commit when they are ready

Case 2a: Incorrect Prediction

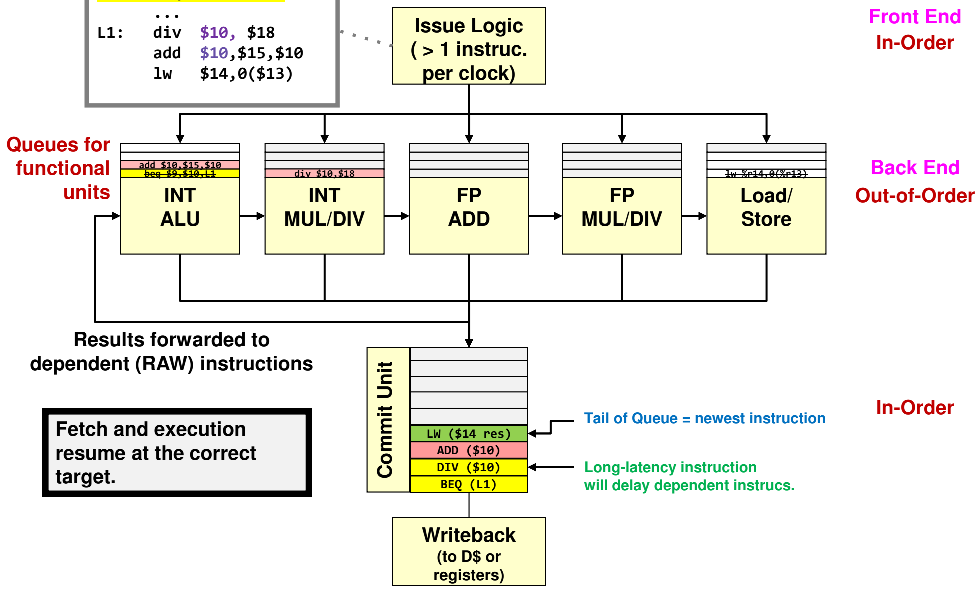
```
        lw   $9,0($8)
        beq  $9,$10,L1
        add  $11,$7,$11
(miss)lw     $12,0($11)
        or   $14,$14,$8
        sub  $12,$14,12
        sw   $12,0($13)
```



If the BEQ is mispredicted, all later (younger) instructions in the ROB and functional unit queues are flushed.

Case 2b: Incorrect Prediction

```
      beq  $9,$10,L1
      ...
L1:   div  $10, $18
      add  $10,$15,$10
      lw   $14,0($13)
```

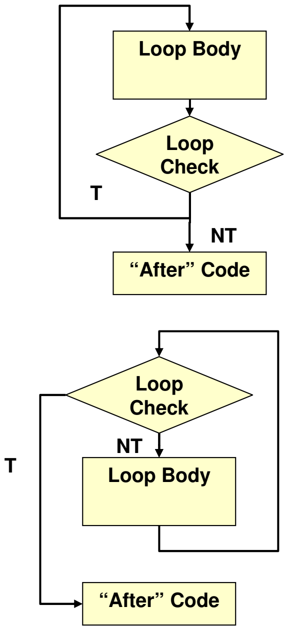


Fetch and execution resume at the correct target.

Making Predictions

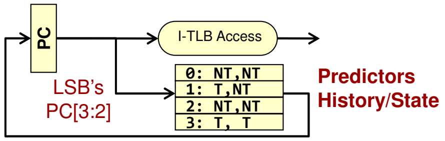
- Many branches have highly predictable behaviors (think loops)
- **Static Predictors**: Generated by the __________ for an ISA that supports prediction hints in the machine code of a branch
- **Dynamic Predictors**: _____ can keep statistics on some number of recent branches to help predict their outcomes

```
int i=50;
do {
  // body
} while (--i != 0)
```



Dynamic Branch Outcome Prediction

- Keep some "history" of branch outcomes and use that to predict the future
- Keep a table indexed by LSB’s of PC with the current prediction
- Questions:
  - What history should we use to predict a branch?
  - How much history should we use/keep to predict a branch?



0: NT,NT
1: T,NT
2: NT,NT
3: T, T

```
0x418 bne $5,$6,L1
      add  $2,$3,$4
      lw   $8,0($5)
0x424 beq  $9,$0,L2
```

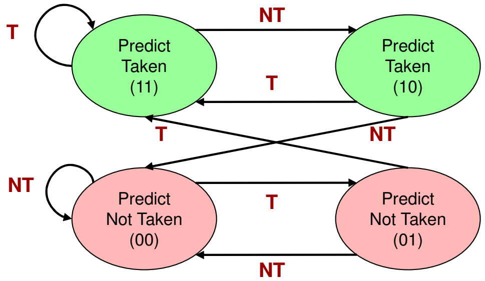
Dynamic Local Predictors

- How much history do we need to keep?
- 1-bit predictor per branch = Last outcome of the branch used to predict next outcome
  - Problem: When wrong once will often be wrong ________
  - Highlighted BNE will be T,T,…T,NT,T,T,…T,NT,T,T,…
    - NT only every 50 iterations
  - 1-bit predictor will say T when bne is NT, then update to NT and be wrong again the next time when bne is T again

```
        addi $a0,$0,10
LOOP1:  addi $a1,$0,50
LOOP2:  ...
        addi $a1,$a1,-1
        bne  $a1,$0,LOOP2
        addi $a0,$a0,-1
        bne  $a0,$0,LOOP1
```

2-bit Predictor

- Solves the problem of 2 mispredictions at the end of a loop
- Keep current prediction (e.g. T) until mispredicted twice in a row (e.g. NT, NT)
  - Require 2 bits for 4 cases of last 2 outcomes
- More than 2-bits does not yield much better accuracy



Assume we start in Predict T state, how many mispredictions will each sequence cause?

					Mispredicts
1.)	T	T	NT	T	1
2.)	NT	T	NT	NT	3
3.)	NT	NT	T	T	4
4.)	NT	NT	NT	NT	2
5.)	NT	T	NT	T	2

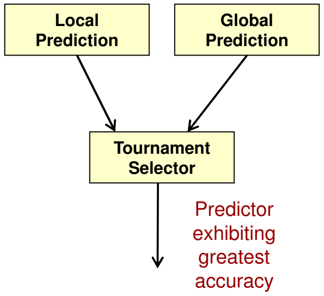
Local vs. Global History

- What history should we look at?
  - Should we look at just the previous executions of only the particular branch we’re currently predicting or at ______________ branches as well
- Local History: The previous outcomes of that branch only
  - Usually good for loop conditions
- Global History: The previous outcomes of the last *m* branches in time (other branches included)

```
do  {
  if(x == 2) { … }
  if(y == 2) { … }
  if(x != y) { … }  // Better:
                    //  Local or Global
}
while (i > 0);       // Better:
                     //  Local or Global
```

Tournament Predictor

- Dynamically selects when to use the global vs. local predictor
  - Accuracy of global vs. local predictor for a branch may vary for different branches
  - Tournament predictor keeps the history of both predictors (global or local) for a branch and then selects the one that is currently the most accurate

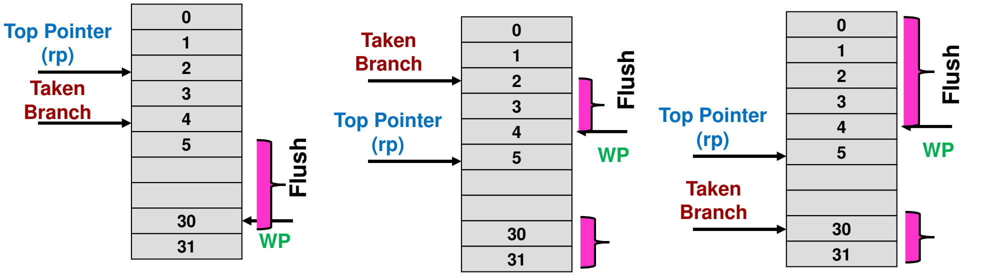


Supporting Speculative Execution

SELECTIVE FLUSHING

Flushing Mechanism

- When we mispredict, we need to flush executed instructions in the _____ and not-yet-executed instructions in the _________________
- To do so, we provide the following to the backend (ROB, Issue queues):
  - A 'flush' command signal
  - Current Top of ROB
  - **Depth** of the Branch Instruction
- All instructions in the backend (as well as the ROB) with depth ________ than the successful branch need to leave (be flushed)



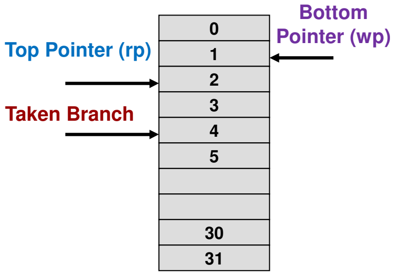
Flush Depth = 2 = (4-2) Flush Depth = 29 = (2-5) mod 32 Flush Depth = 25 = (30-5) mod 32

Selective Flushing for Branch Misprediction

- Paper token analogy
  - Say the store is going to close in 20 min. and they noticed too many people are waiting
  - They may announce that they will serve up to token #72 and people having tokens after that may leave now
- If the last token pulled is 92, then people with tokens #73 to #92 will leave
- If the last token pulled is #32, then people with tokens ____________ and _____________ will leave
- Because of the circular nature of the tokens/ROB FIFO mechanism, one cannot simply compare his token with #72 to decide whether to stay or leave
- Leave if you are more than 20 people away from current person being served (i.e. #52)

Selective Flushing for Branch Misprediction

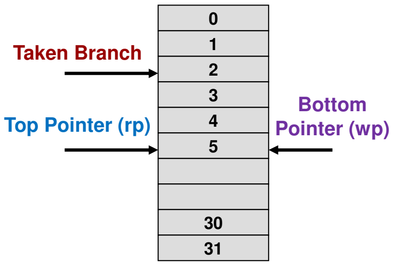
- Anyone with greater depth (distance from top pointer) than the branch should leave
- Suppose the bottom (WP) is at 1
  - Is it (ROB) full? Yes / No
  - Total Populated Area = 1 / 31 / 32
- Who should leave (be flushed)?
  - Those with distance
  - Note: #1 is empty



Depth = 2 = (4-2)

Selective Flushing for Branch Misprediction

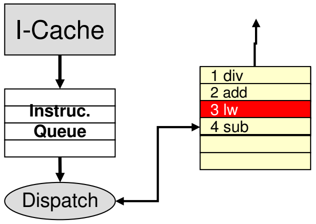
- Who should leave in this scenario?



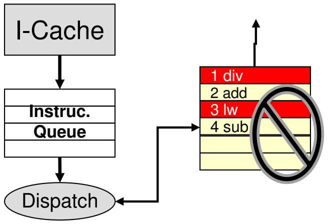
Flush Depth = 29 = (2-5) mod 32

Precise Exceptions

- Only handle exceptions from an instruction that is at the head of the ROB
  - It may have detected the exception case while executing but stored necessary info in the ROB and _____________ until it reached the HEAD to actually generate the exception
  - Flush all instructions in the ROB and restart fetching from the exception handler
- Supports PRECISE exception model!



LW detects exception but waits to become head of ROB



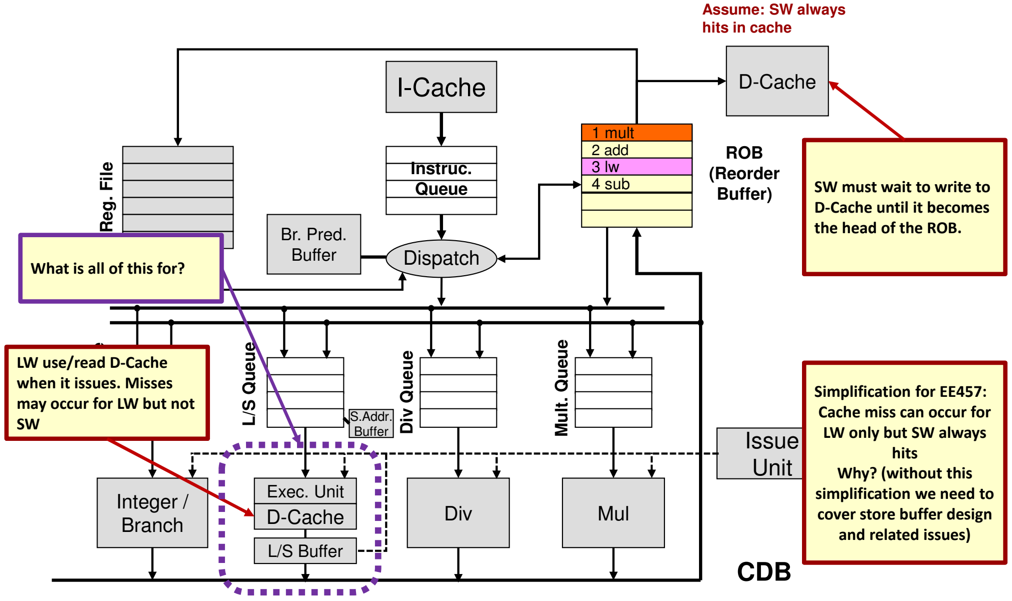
Meanwhile if DIV then generates exception, take it and flush ROB

MEMORY DISAMBIGUATION

Register Hazard Summary

- Recall, RAW hazard for registers was handled by
  - Dependent instructions are given the ROB tag of their specific producer to wait on in the backend
  - When the specific producer comes on the CDB an announces the value, then the dependent instruction grabs the value
  - Once the dependent instruction has all its sources, it raises his hand to say, "I am ready to go the execution unit" and waits for the issue unit to grant permission
- We must still take care with WAR and WAW hazards for registers, but we do so by taking ROB tags (solves WAR) and In-Order Completion/Writeback (solves WAW)

Tomasulo 2: Memory Assumptions



Assume: SW always hits in cache

SW must wait to write to D-Cache until it becomes the head of the ROB.

What is all of this for?

LW use/read D-Cache when it issues. Misses may occur for LW but not SW

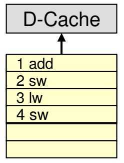
Simplification for EE457: Cache miss can occur for LW only but SW always hits
Why? (without this simplification we need to cover store buffer design and related issues)

RAW, WAR, WAW for Memory

- We said hazards may occur in memory
- WAR and WAW hazards are handled through In-Order Completion
  - R = Read = LW (load word)
  - W = Write = SW (store word)
- An 'LW' reads cache in the execution unit before going to ROB
- An 'SW' writes into cache (i.e. commits) when it reaches the "top" of ROB (meaning it became the oldest instruction)

```
// Dependency?
SW $2,0($5)
LW  $8,0($5)

// Dependency?
SW $2, 1000($4)
LW $3, 0($6)
```



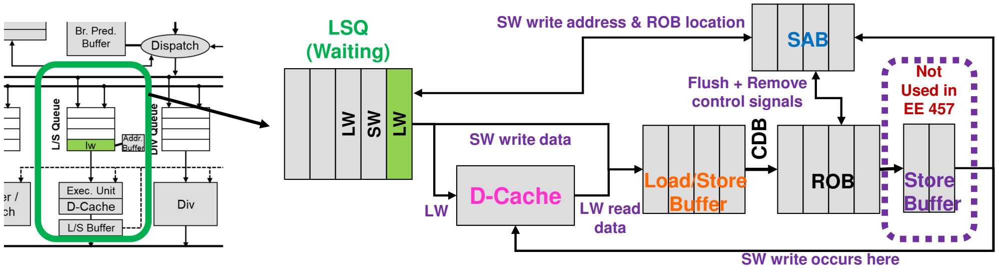
ROB

Reorder Buffer and In-order Completion solve WAW (and helps with WAR)

Meet the Components

Load Store Queue (LSQ) – Holds Loads and Stores until they have their requisite source operands and issues them.

Store Address Buffer (SAB) – Holds pending stores addresses and ROB tags to help solve memory hazards (disambiguation) and allow LWs in the LSQ to issue as early as possible



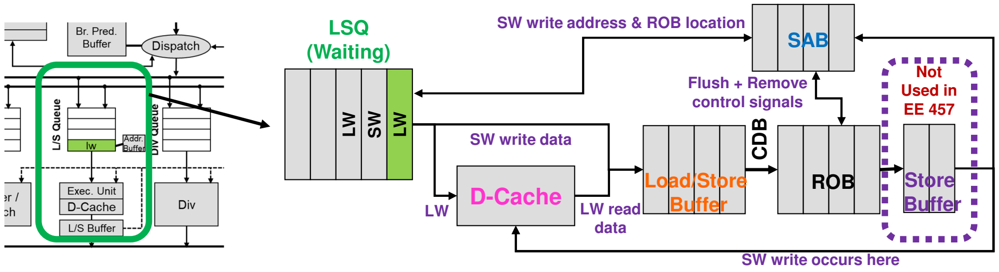
Cache – Assume misses are allowed for LW but we assume all SWs hit (to avoid the need for the Store Buffer)

Load/Store Buffer (LSB) – Since latency of a LW is unknown (due to cache miss), this aides scheduling for writing to the ROB over the CDB. Stores also use it to give a common I/F to the ROB

Store Buffer (SB) – Not Used in EE 457 – Helps if SW misses in cache to allow ROB to keep committing junior instructions

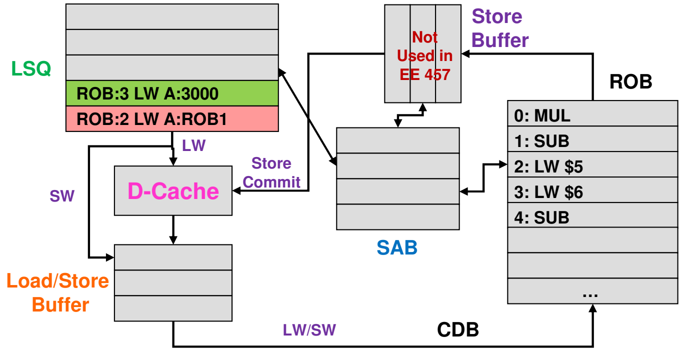
A Few More Notes

- LW accesses cache and result is written into **Load/Store Buffer**
- SW does not access memory while getting issued from LSQ and goes to **SAB** (see next bullet)and the **Load/Store buffer** directly, then on to the ROB
- Whenever SW issues, its write address and ROB location are stored in the **Store Address Buffer** (for fast detection of latest SW before a LW)
- Once an SW is committed from the top of the ROB its entry in the **Store Address Buffer** is cleared



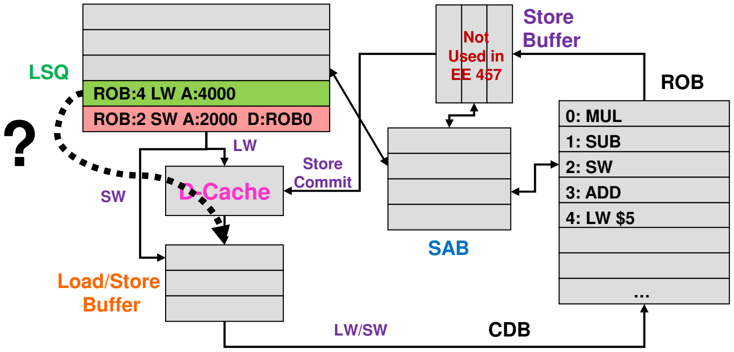
LW Issue (1)

- Can LW (ROB2) issue/execute?
  - ____. Must know its __________.
- Can LW (ROB3) skip ahead of LW (ROB2) and issue/execute?
  - _______, LW can skip other _______ even with __________ addresses



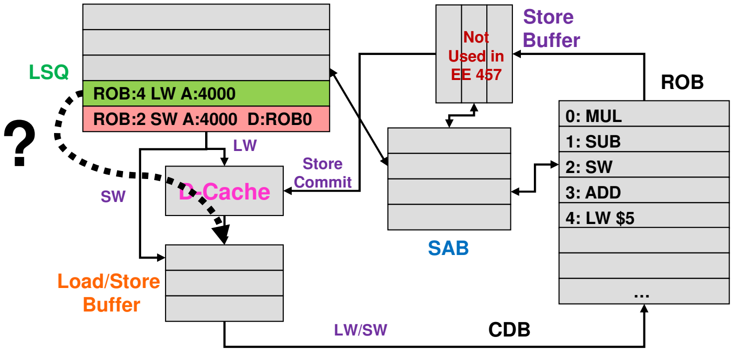
LW Issue (2)

- Can LW (ROB4) skip ahead of SW (ROB2) and issue/execute?
  - ______, as long as the addresses are ____________, LW can issue ahead of a SW



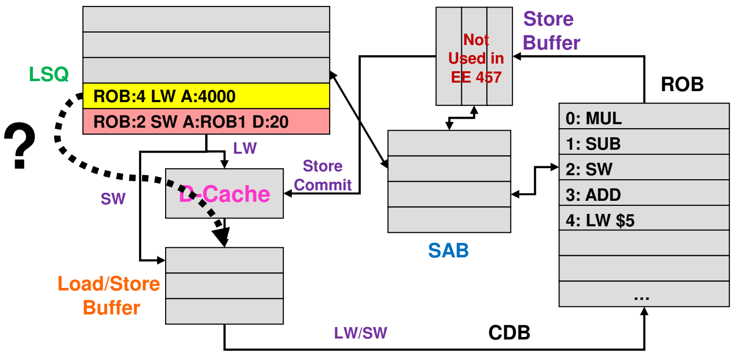
LW Issue (3)

- Can LW (ROB4) skip ahead of SW (ROB2) and issue/execute?
  - ______, SW address ____________ LW's address creating a _______ hazard we must respect



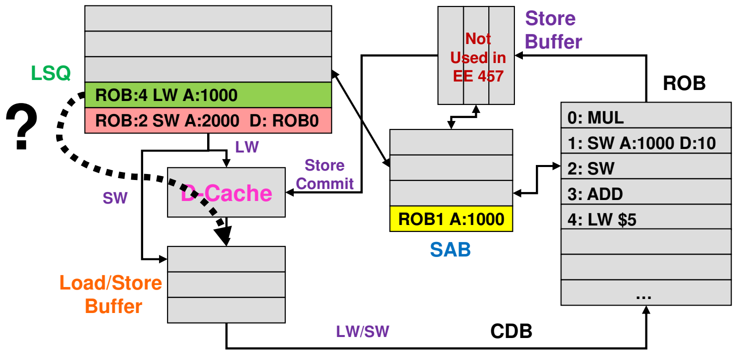
LW Issue (4)

- Can LW (ROB4) skip ahead of SW (ROB2) and issue/execute?
  - _________. What if SW address ends up being _________ which is a ________ hazard.



LW Issue (5)

- Can LW (ROB4) skip issue/execute?
  - No, a SW with the ________ address is ahead of us in the ROB (and SAB also records the address to help us search quickly) and has NOT written yet. We must wait for it to write when it reaches head of ROB.
  - Could potentially grab data from ROB but this makes the design more _____________, though can and is done in most OoO processors.



LW Issuing

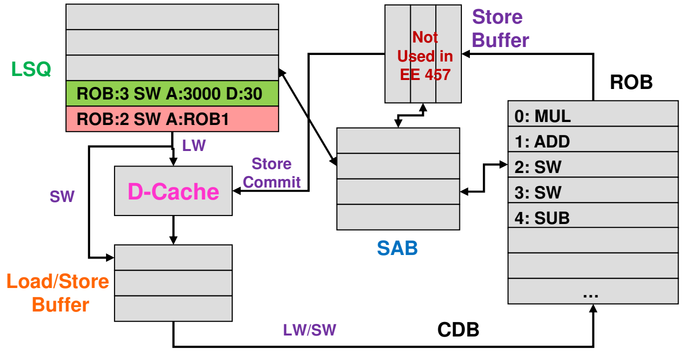
- To handle RAW properly an LW must wait in LSQ until:
  - It knows its read address
  - All senior SWs know their write address (SW may be waiting on some earlier instruction for its write address)
- Then either
  - Wait and read data from cache once no earlier (older) SW's are in the LSQ or Store buffer ...OR...
  - [Not in EE 457] Get data directly from prior SW (latest of those SW's with matching addresses) out of the Store buffer after the SW had reached the head of the ROB
- We use the "Store Address Buffer" to maintain a record of SW addresses and perform fast comparisons to help waiting LWs determine if there are older SWs in the ROB, Store buffer, etc. and to aide prioritization to find matches and the "youngest" of the "oldest"

```
// RAW Mem Hazard

SW $2, 1000($4)
LW $3, 0($6)
```

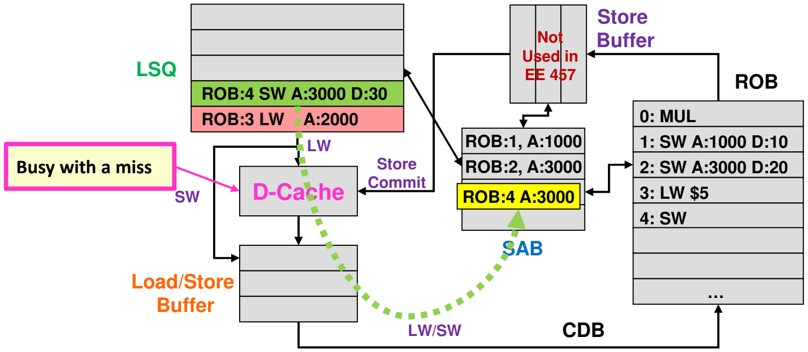
SW Issue (1)

- Can SW (ROB3) issue/execute ahead of SW (ROB2)?
  - _______, even though SW (ROB2) may end up with the same address as SW (ROB3), they only go to the ______ and don't write to memory until they reach the _________ of ROB. This means they will write _________________ regardless of when the "issue"/"execute".



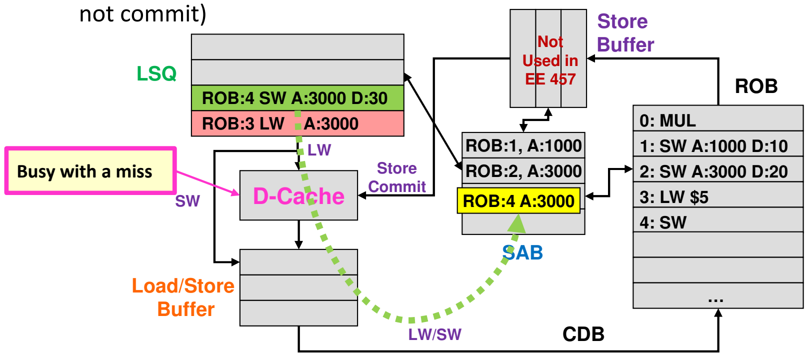
SW Issue (2a)

- Assume cache is busy with a miss, can SW (ROB4) issue/execute ahead of LW (ROB3)?
  - ________, since their addresses are _______________



SW Issue (2b)

- Can SW (ROB4) issue/execute ahead of LW (ROB3) with the same address?
- ____, this would then hold off the LW from issuing/executing when it sees the match in the SAB (i.e. it will think there is a RAW hazard when its actually a _______ hazard)
- Furthermore, if _____________ address matches in SAB how would LW know whom to get data from? [At the very least it complicates determining it.]
- Could lead to _______________ if it's waiting on a SW after it but that means that SW will never reach the head of the ROB because LW doesn't execute (and thus does not commit)

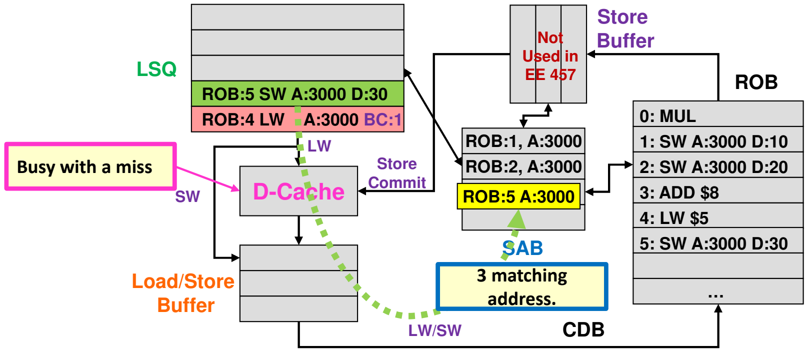


SW Issuing and Bypassing

- But with a little thinking, we can, in fact, allow _________ SWs to bypass a waiting earlier LW by making the LW keep a count (aka __________ **count**) of how many bypassing SWs matched its address
- When LW can issue when the number of matches in the SAB equal the number of SWs that bypassed it

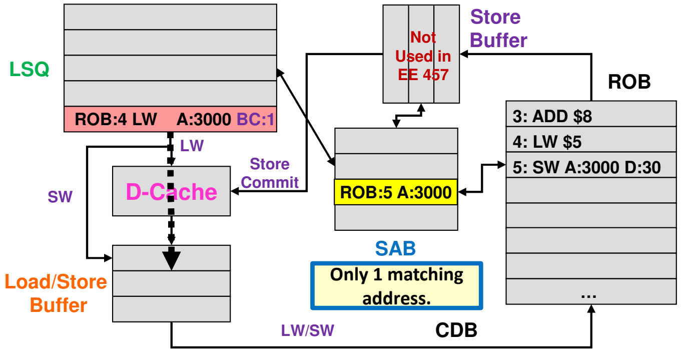
SW Issue (3a)

- Now we add a **bypass counter (BC)** to the LSQ entries for LWs
- When a later (junior) SW bypasses (skips ahead of) a LW that maches its address, the LW increments its bypass counter (BC)
- Notice currently, LW has two SWs with matching address in front of it and thus must wait for them to write.



SW Issue (3b)

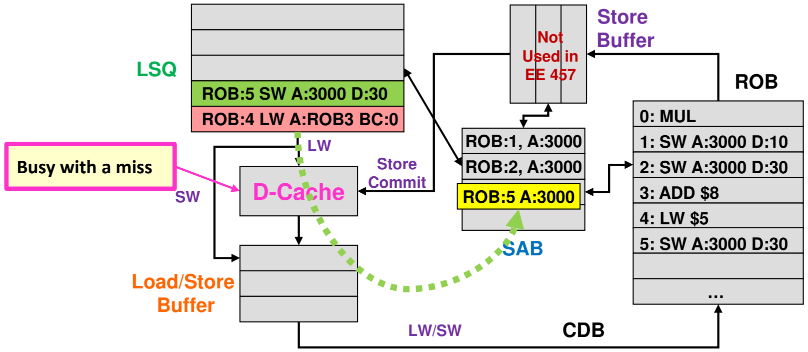
- Once the SWs that were ahead of the LW in the ROB commit (and their SAB entries invalidated), the LW can see there is 1 matching address in the SAB which is ________________ to its bypass counter.
- Thus, the LW can and should execute



Only 1 matching address.

SW Issue (4)

- Can SW (ROB5) bypass the LW in this example?
  - ______. Since the LW's address is _________. If the SW jumps ahead, the LW does not know if it should or should NOT increment its bypass counter, leading to incorrect behavior
  - SW can only issue over a LW that ___________ its address



Busy with a miss

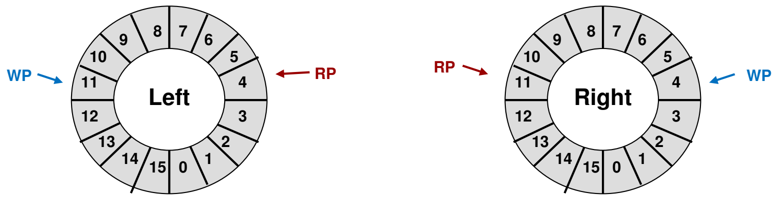
Store Word Issuing

- An SW can issue when its
  - Write address is known
  - Write data is known
  - No LW in front of it has an unknown address
    - Because LW won't be able to keep track of the count of how many matching SW's bypassed it

SOME OLD REVIEW PROBLEMS

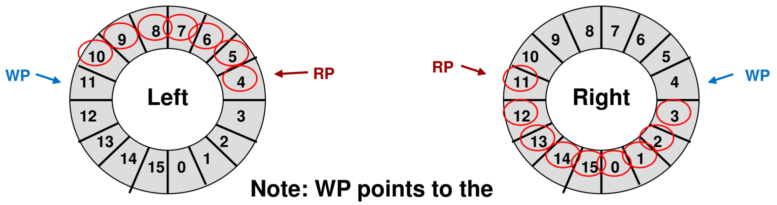
Spring 2011 Final Exam Question

- In the illustrations below, the ROB is (more / less) than half-full in the left case and is (more / less) than half-full in the right case. The left has ____ locations occupied and the right has ___ locations occupied. WP is (always / sometimes / never) ahead of RP.



Spring 2011 Final Exam Solution

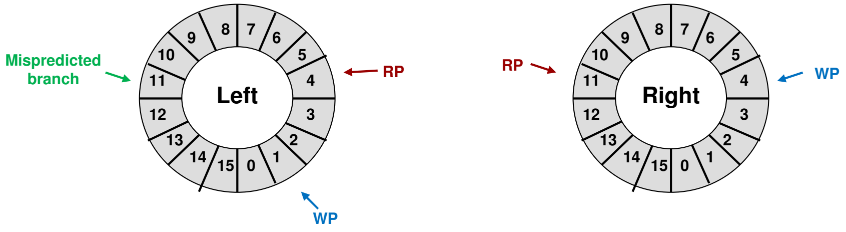
- In the illustrations below, the ROB is (more / less) than half-full in the left case and is (more / less) than half-full in the right case. The left has 7 locations occupied and the right has 9 locations occupied. WP is (always / sometimes / never) ahead of RP.
  - Except on reset when WP=RP



Note: WP points to the location yet to be written

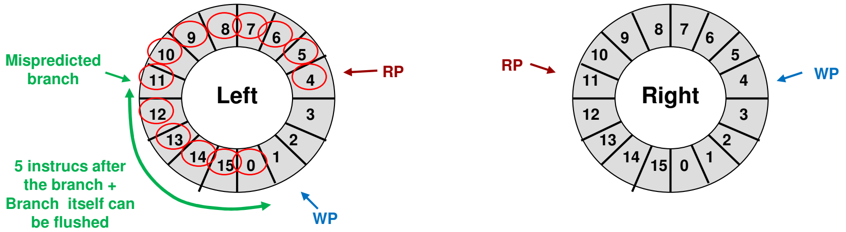
Spring 2011 Final Exam Question

- If the instruction with ROB Tag 11 is found to be a mispredicted branch, which instructions with what ROB tags would you flush?
- And would you adjust RP or WP or both? And to what value(s)?



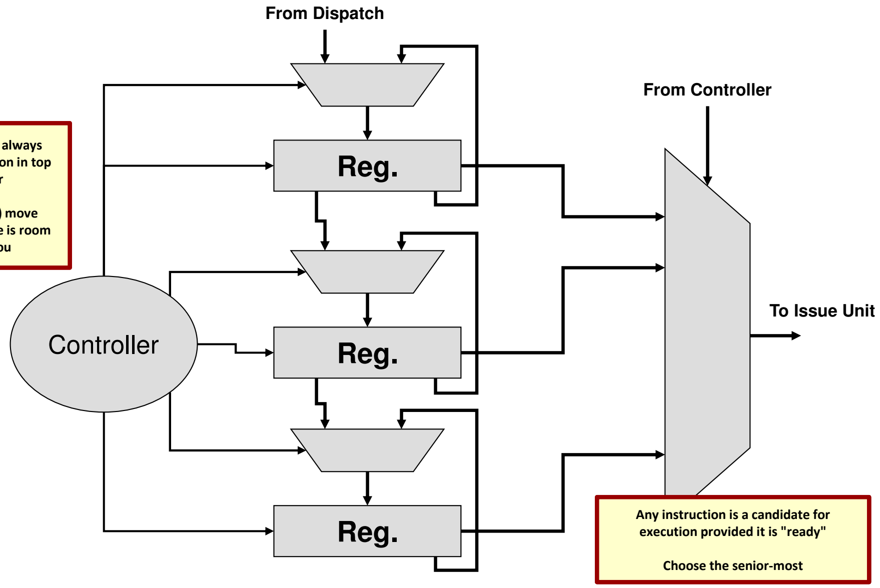
Spring 2011 Final Exam Solution

- If the instruction with ROB Tag 11 is found to be a mispredicted branch, which instructions with what ROB tags would you flush?
  - Instructions with ROB tags 12, 13, 14, 15, and 0 should be flushed as they are younger than the branch
- And would you adjust RP or WP or both? And to what value(s)?
  - We will adjust WP to 11



OTHER IMPLEMENTATION DETAILS

Issue Queues



Dispatch Unit always places instruction in top register

Instruction(s) move forward if there is room below you

Any instruction is a candidate for execution provided it is "ready"

Choose the senior-most