

Lesson 7B

ROB
+ RAT (Register Renaming) - Gets rid of false dependencies
+ Tomasulo's - OoO Execution
+ ROB - In-order COMMITS

ReOrder Buffer Rev. 2

The ReOrder buffer is found in all modern high performance processors. It is used to ensure correct program outcomes in out-of-order processors.

Exceptions in Out-of-Order Execution

The major drawback of Tomasulo's algorithm is that it does not handle exceptions very well. When an exception occurs in OOO processors the instructions after the exception producing instruction have been executed, resulting in incorrect outcomes.

Branch Mispredictions in OOO Execution

Tomasulo's algorithm does not handle branch mispredictions in OOO processors well.

Phantom Exceptions - if a branch is mispredicted a number of instructions are ^{incorrectly} executed before the misprediction is detected. It is possible to generate an exception within these executed instructions, now the processor must recover from an exception that should not have occurred (in the first place) as well as the misprediction.

Exception handling should not occur until the processor is sure they are not phantom exceptions.

Correct OOO Execution

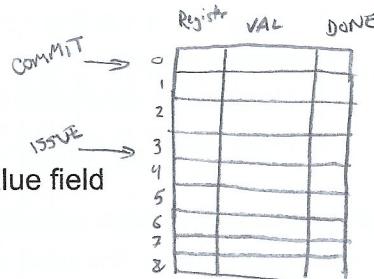
- Execute OOO
- Broadcast OOO
- Deposit values in registers In-Order. This can be achieved with a ReOrder Buffer.

A ReOrder Buffer remembers the program order and keeps the results until it is safe to write them.

ROB Part 1

The ROB has three fields

- The data being held by the ROB
- Done bit, which tracks the validity of the data in the value field
- Which register is holding the data



Instructions are kept in program order.

There are two pointers associated with the ROB:

- Commit tells which instruction is to be completed next
- Issue tells where the next issued instruction is to be placed in the ROB

ROB Part 2

How the ROB interacts with the IQ, RS, Regs, and the RAT.

ON INIT, both the ISSUE + COMMIT ptrs are pointing to the same ROB entry.

Do this by writing down the result register in the ROB entry.

The steps for Issue when using a ROB

1. Get the instruction from the Instruction Queue
2. Get a free Reservation Station
3. Get the next ReOrder Buffer entry (at the Issue pointer)
4. Look up the source registers in the RAT (RAT points to a ROB entry or Register)
5. Change the RAT for destination registers to point to the ROB entry instead of the RS (which we did in Tomasulo's)

Populate the entry + then increment the ISSUE ptr.

Dispatch with the ReOrder Buffer

Steps for dispatch when using a ROB

- same as Tomasulo's -
1. Find results where the tags match. (+ capture those results)
 2. Only instructions where all the inputs are ready are considered for dispatch
 3. Pick one instruction for each functional unit
 4. Free the reservation station. This is different than Tomasulo's algorithm. With a ROB the instruction is broadcast with the ROB name rather than the RS name, so the RS can be freed sooner.
- in Tomasulo's, we needed to keep the RS occupied until the results were broadcast (be it other instructions were waiting for that result + we freed it + another instruction took it, then the waiting instruction could get it)

ROB 3

Broadcast with the ReOrder Buffer

← Same as Tomasulo's, except tag that we're broadcasting with is the Rob entry, not the RS

The steps for broadcasting with a ROB:

1. Capture the result in the waiting Reservation Station
2. Write the result to the ROB (unlike Tomasulo's, where we'd write DIRECTLY to the register file)

DO NOT UPDATE THE RAT (you'd do this in Tomasulo's)

Commit with the ReOrder Buffer

The steps to commit with a ROB

1. Look at only the oldest instruction in the ROB
2. Wait for this instruction to complete (DONE bit = 1)
3. Copy the result from the ROB to the register. This means that register writes are occurring in order.
4. Update the RAT with the destination register- (if there's a 'RAT entry that has the ROB entry tag, then clear it out to indicate that the latest value is in the register file. If there's no matching one, then don't touch the RAT. Look at "RAT updates on commit" on the next page!)
5. Free the ROB entry (clear "Done" bit + increment "COMMIT" ptr)

Review of how Tomasulo's broadcast works

Our result would be tagged w/ the RS #, & if the RAT had an entry mapping a register to that RS # tag, THEN we'd update the actual register. Otherwise, if there's no matching RAT entry w/ that RS# tag, we don't touch the regis

Hardware Organization with the ROB

The ROB has a head, the next instruction to be issued, and a tail, the next instruction to commit. Entries in between these two are being executed.

The ROB is used to:

- Remember the program order
- Temporarily store an instruction's result
- Serve as the name (tag) for the result

For instructions that don't write a result to a register (like a branch instruction), they still take up a ROB entry.

Reg	Val	Done
RI	/	
/	/	
/	/	

ex: LD R1, \$ (R1)
BNZ R1, R2, Label

if BNE was a misprediction then it's corresponding ROB entry VAL would store that result (Val = !) same for exceptions

Branch Misprediction Recovery Part 1

Branches are not committed until they are resolved. This means instructions that come after the branch are also not committed until the branch is committed.

So branch misprediction recovery requires a flush of the ROB, RS, and RAT, then the correct instructions are fetched from the correct address. The ROB is flushed simply by moving the Issue pointer to the same entry as the Commit pointer. The RAT entries are set to point to the correct registers. (aha they're cleared out)

ROB and Exceptions

Two problems with exceptions:

1. OOO instructions - When an exception occurs the ROB has not committed, so everything can be flushed and the exception handler can be loaded.
2. Phantom Exceptions - Since nothing is committed after the branch, any misprediction results in a flush of the instructions, including any exceptions, and a load of the correct instructions.

Treat the exception the same as any other result → delay the actual handling of the instruction exception until the commit.

Committed instructions cannot be "uncommitted".

RAT Updates on Commit

- Commit instruction
- The result is put in the registers ALWAYS - use the "Register" field to tell you which Register to update
- Check the RAT. If the RAT entry is not the latest RAT update, nothing needs to be done to the RAT. If it is the latest update, change the value in the RAT to correspond to the correct register.

Registers are always up-to-date, which improves exception handling.

The values are updated on commit only if renaming occurs.

Unified Reservation Station

Unified reservation stations: combine the reservation stations for the Adder and the Multiplier. All of the reservation stations are in one large array, so the reservation stations can be used when necessary, they do not have to wait for a specific set of RS.

The down side of the unified RS is the increased complexity in hardware. The hardware must be able to determine the correct arithmetic unit for each entry.

Superscalar

A superscalar processor must be able to:

- Fetch more than 1 instruction per cycle
- Decode more than 1 instruction per cycle

- Issue more than 1 instruction per cycle
- Dispatch more than 1 instruction per cycle
- Broadcast more than 1 instruction per cycle
- Commit more than 1 instruction per cycle

Weakest link: the processor is limited by the slowest task. If all of the tasks, except Decode, can perform 4 instructions per cycle and Decode can only perform 2 instructions per cycle, then the processor is limited to 2 instructions per cycle.

Terminology Confusion

Most research papers will use:

Issue, Dispatch, Commit

Processor Designers will often give these different names

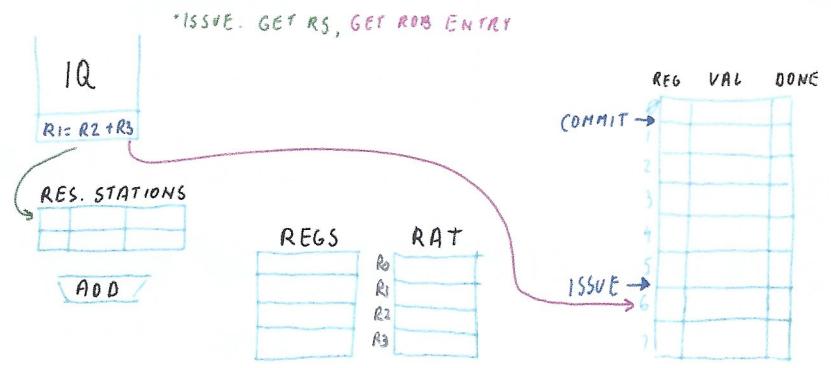
Out of Order

What is really out of order?

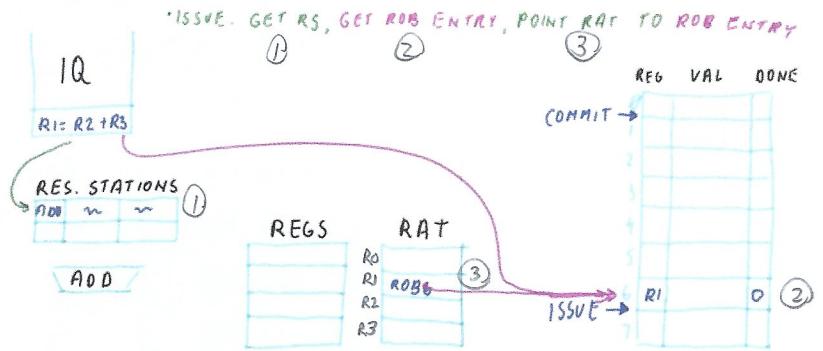
Fetch, Decode, Issue are all in order - this ensures dependencies are done in order

Execute and Write can be done out of order

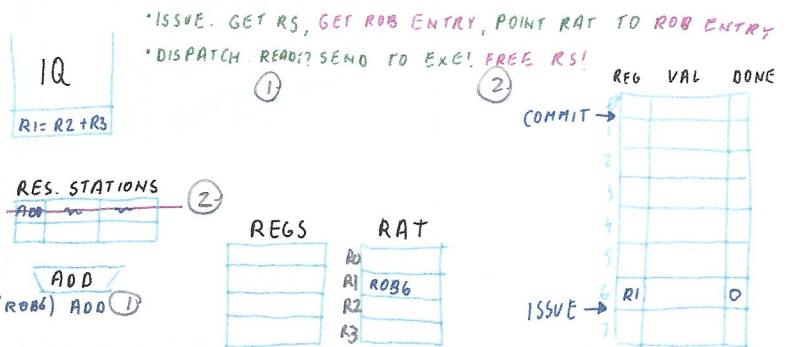
Commit is done in-order



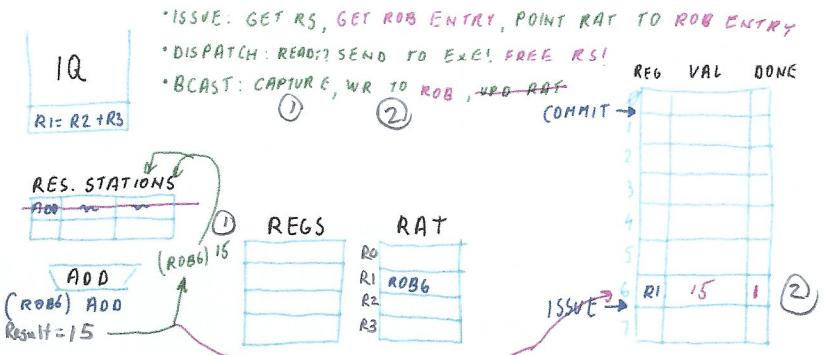
Initial:



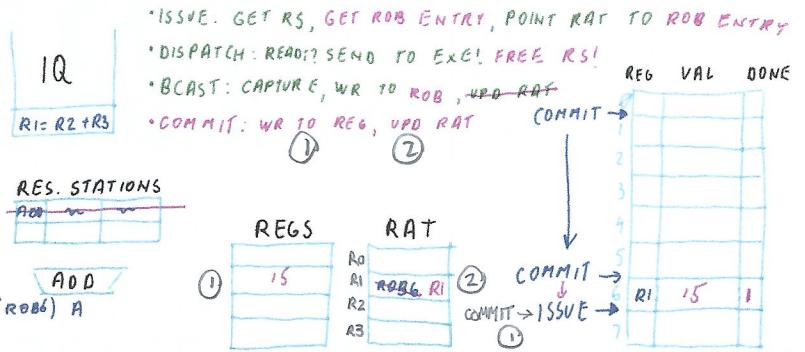
Issue:



Dispatch:



Broadcast:



Capture:

ROB TIMING EXAMPLE

- FREE RS ON BCAST, NOT DISP
- ISSUE, CAPTURE, DISPATCH. IN SAME CYCLE
- 2 MUL RS
- 3 ADD RS

Remember: a dependent instruction in a RS can get the result if it is waiting on during the "Write Back" stage (NOT COMMIT). Then it can execute if it has all the values it needs.

Add: 1 cycles
Mult: 10 cycles
Divide: 40 cycles

Inst	Operands	Is	Exec	Wr	Commit	Comments
DIV	R2, R3, R4	1	2	42	43	
MUL	R1, R5, R6	2	3	13	44	
ADD	R3, R7, R8	3	4	5	45	
MUL	R1, R1, R3	14	15	25	46	NEED RS → ISSUE
SUB	R4, R1, R5	15	26	27	47	EXE DEP ON R1
ADD	R1, R4, R2	16	43	44	48	EXE DEP ON R2

ROB TIMING QUIZ SOLUTION

- BCAST ONE ADD/SUB AND ONE MUL/DIV PER CYCLE
- COMMIT UP TO 2 INSTS/CYCLE
- FREE RS AT DISPATCH

Add: 1 cycles
Mul: 2 cycles
Div: 4 cycles

Inst	Operands	Is	Exec	Wr	Commit	Comments
DIV	R2, R3, R4	1	2	6	7	
MUL	R1, R5, R6	2	3	5	7	
ADD	R3, R7, R8	3	4	5	8	
MUL	R1, R1, R2	4	7	9	10	
SUB	R4, R2, R5	5	7	8	10	
ADD	R1, R4, R3	6	9	10	11	COMMIT 11