High Performance Computer Architecture (CS60003)

Dept. of Computer Science & Engineering
Indian Institute of Technology Kharagpur

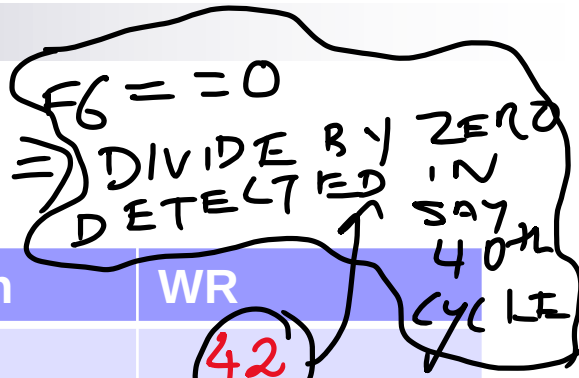
Spring 2023

Reorder Buffer

Exceptions in OOO Execution

S NO	Inst	IS	Dispatch	WR
1	DIV F10,F0,F6	1	2	42
2	LD F2,45(R3)	2	3	23
3	MUL F0,F2,F4	3	14	19
4	SUB F8,F2,F6	4	14	15

Exceptions in OOO Execution



S NO	Inst	IS	Dispatch	WR
1	DIV F10,F0,F6	1	2	42
2	LD F2,45(R3)	2	3	13
3	MUL F0,F2,F4	3	14	19
4	SUB F8,F2,F6	4	14	15

What we should have now:

1. PC for this instruction should be saved.
2. We should jump to an exception handler. The exception handler can try to fix by say making F6 a very small value.
3. If the exception handler comes back, we would be back to this instruction and start executing from here.

What we have:

1. We have executed 2, 3, 4, and only 1 has not been executed.
2. If we now go to the exception handler and come back, instruction 1 will use F0 produced by Instruction 3 and thus will never get the correct result.

Exceptions in OOO Execution

- Can also happen if the load instruction encounters a page fault.
- By the time, the data comes from the disk, the following instructions have already executed.
- So, what we want is precise exceptions:
  - An exception for which the pipeline can be stopped, so instructions that preceded the faulting instruction can complete, and subsequent instructions can be flushed and redispatched after exception handling has completed.

Branch Misprediction

S NO	Inst
1	DIV R1, R3, R4 (40 cyces)
2	BEQ R1, R2, label
3	ADD R3, R4, R5

1. Once DIV is computed we can resolve the branch.
2. So, it would take several clock cycles to realize that the branch has been mis-predicted.
3. If meanwhile, I3 is executed, R3 is already written!
4. Ideally once we realize the misprediction, we should behave as if I3 never executed, and start fetching instructions from ‘label’ that the branch was pointing to.
5. But here (in the ideal Tomasulo’s architecture), by that time R3 could already change.
    1. So, instead of the old value of R3, now we have a new and erroneous value of R3.

Phantom Exceptions

S NO	Inst
1	DIV R1, R3, R4 (40 cyces)
2	BEQ R1, R2, label
3	ADD R3, R4, R5
4	DIV …

Again suppose, the branch was mis-predicted as non branch.

Thus the following ADD and the 2nd DIV instructions are wrongly executed.

Now if the 2nd DIV raises an exception, it is called a phantom exception. When we come back after realizing the mis-prediction of the branch, it is too late:

the 2nd DIV already would have raised an exception handler before knowing it should not have occurred!

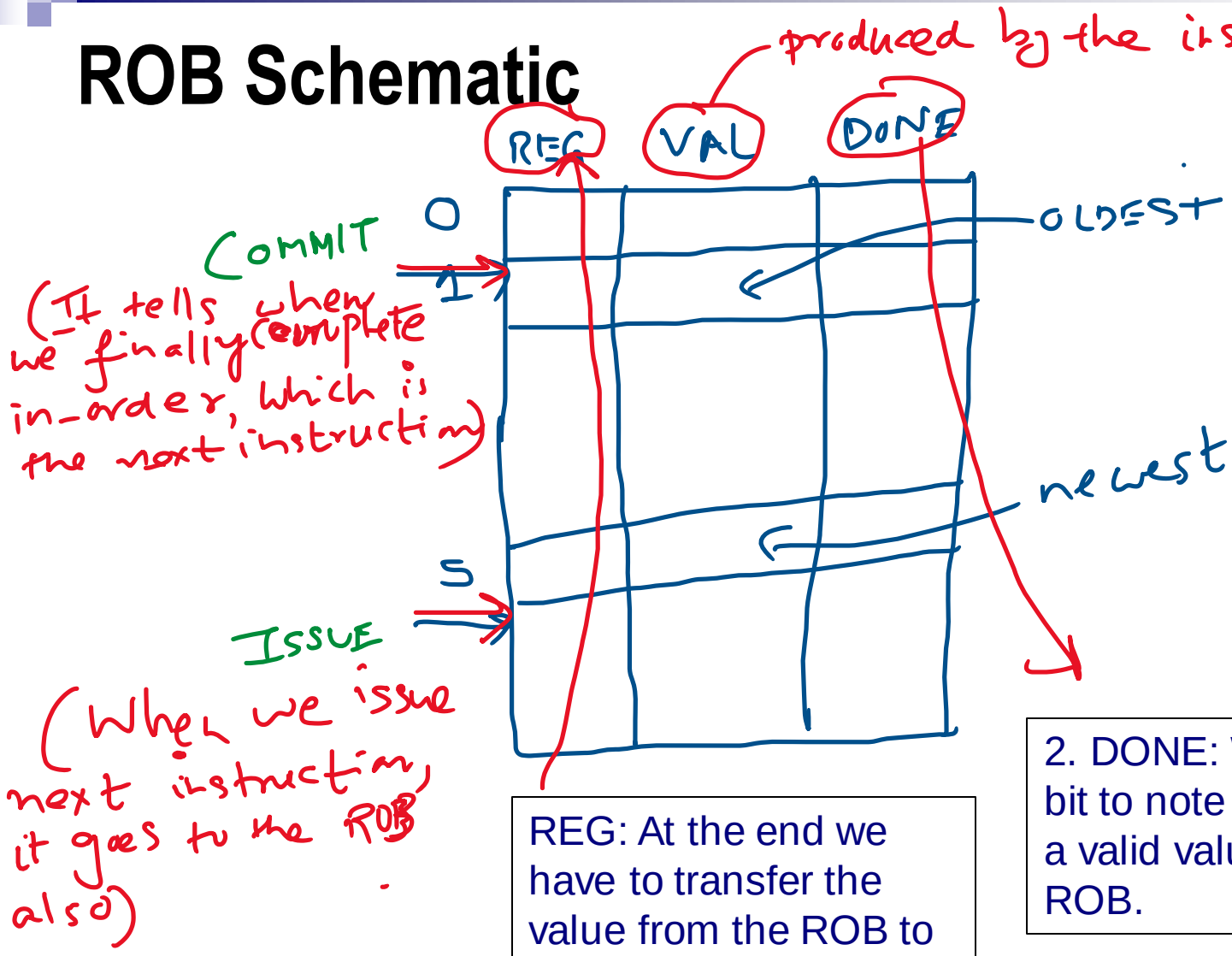
Correct OOO execution

- Execute OOO
- Broadcast OOO
- **But write to registers in-order!**
  - This is because if values are deposited OOO, then if we discover that one of the previous instructions should not occur (because of exceptions, or branch-misprediction), then if an OOO executed instruction has written into the registers, we have a problem!
  - But if we deposit in-order, then it implies all the previous instructions have finished execution, and all is well!

Reorder Buffer (ROB)

- We use a unit called ROB.
- Even after issue, reorder buffer:
  - Remembers program order
  - Keeps result until it is safe to write
- So, basically rather than writing to registers immediately as the result is ascertained, we would store it in the ROB.
- Then we process the ROB in-order and deposit the results in the register.
- Only then the instruction is considered fully done, which we indicate in the ROB also.

ROB Schematic



VAL: This is the value that we are not writing to the register. When the instruction finishes and broadcasts, it will write its value to the ROB.

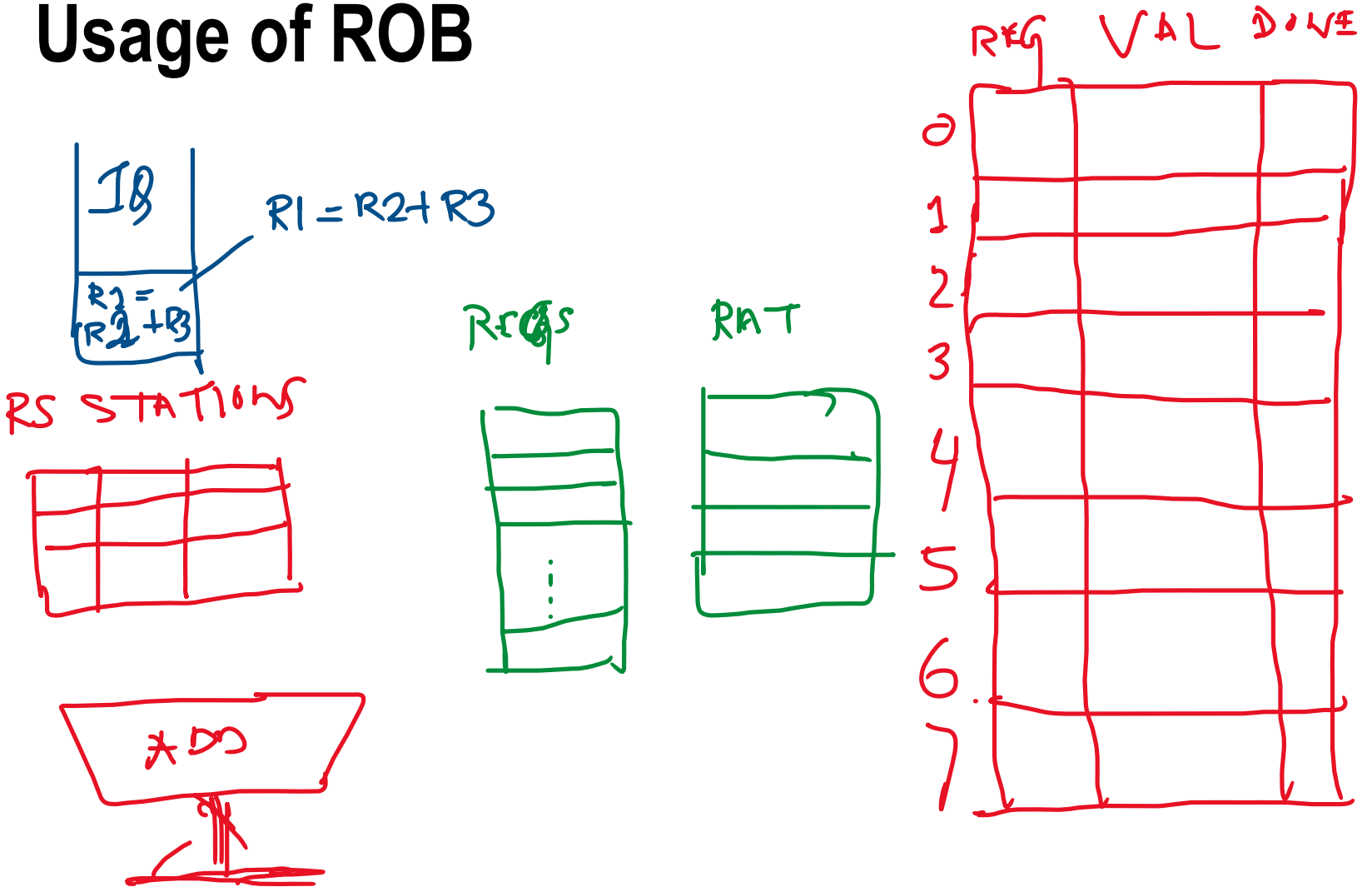
2. DONE: We need a bit to note that we have a valid value in the ROB.

REG: At the end we have to transfer the value from the ROB to the register file, for which we need the REG field.

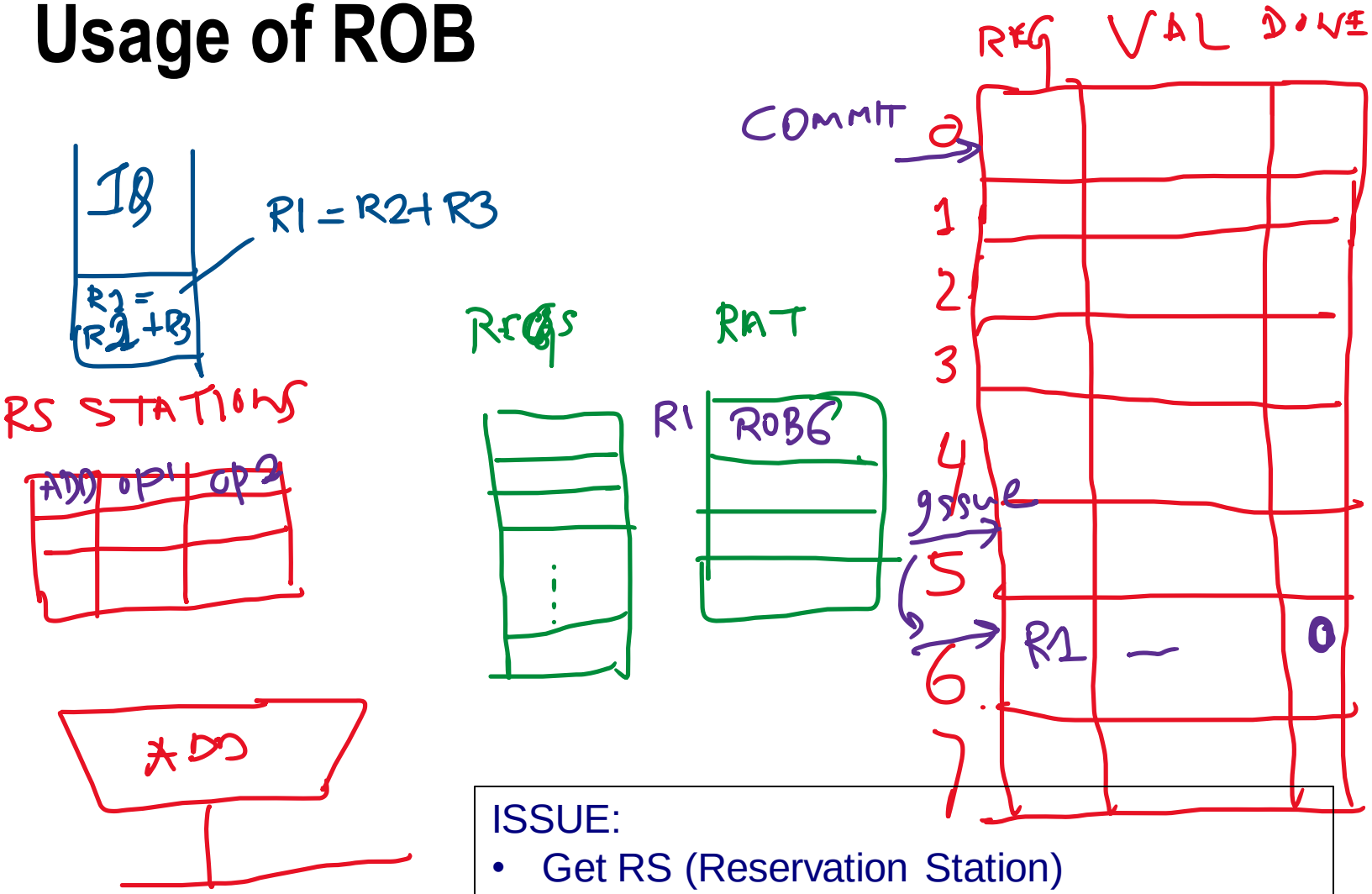
ROB

- It remembers the program-order.
- It temporarily stores instruction result.
- It serves as a name (tag) for the result.

Usage of ROB


IB
R1 = R2 + R3
R1 = R2 + R3
RS STATIONS
REGS
RAT
ADD
REG
VAL
DIRE
0
1
2
3
4
5
6
7

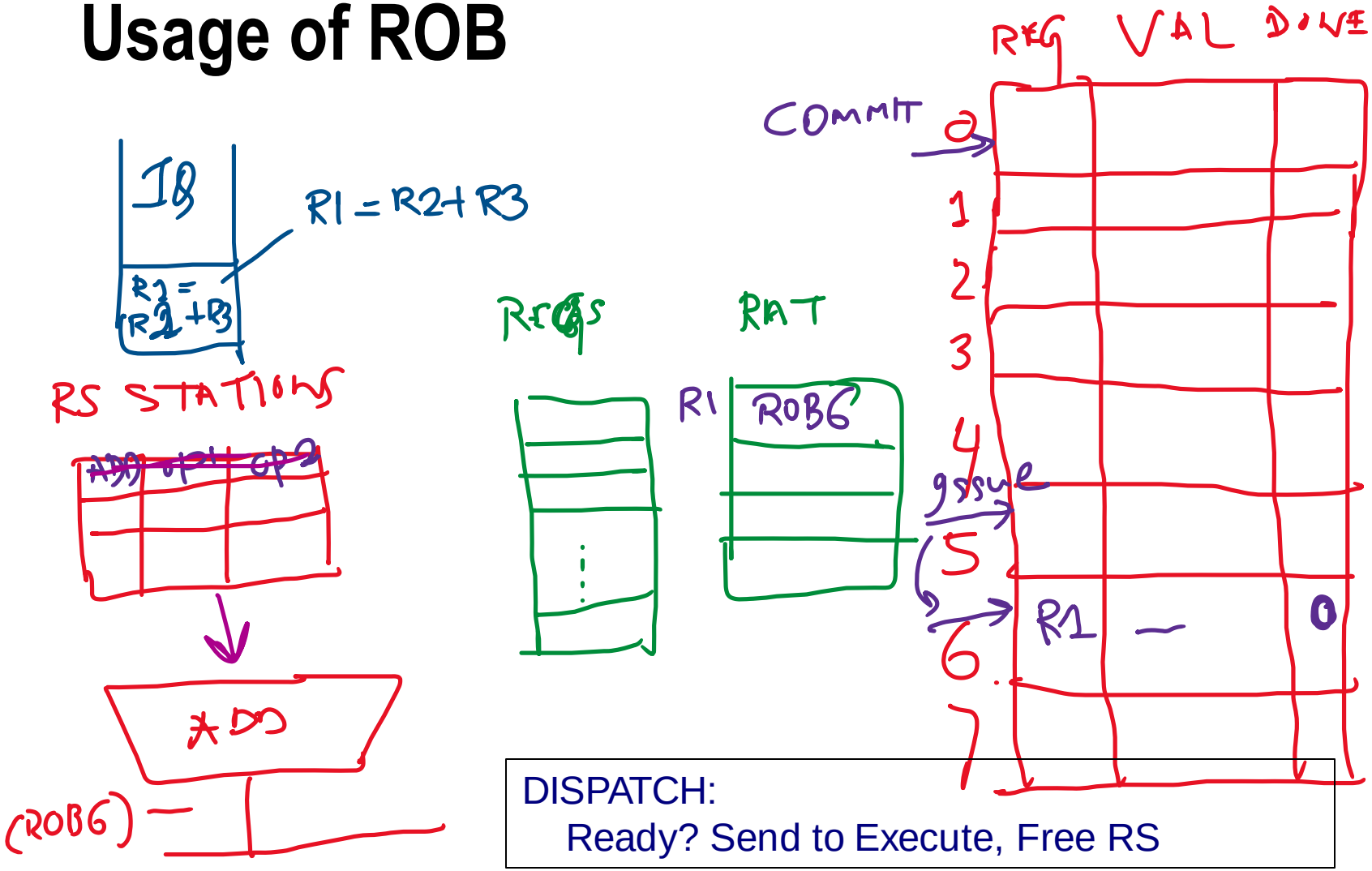
Usage of ROB



ISSUE:

- Get RS (Reservation Station)
- Get ROB entry.
- Point RAT to the ROB entry (not RS entry)

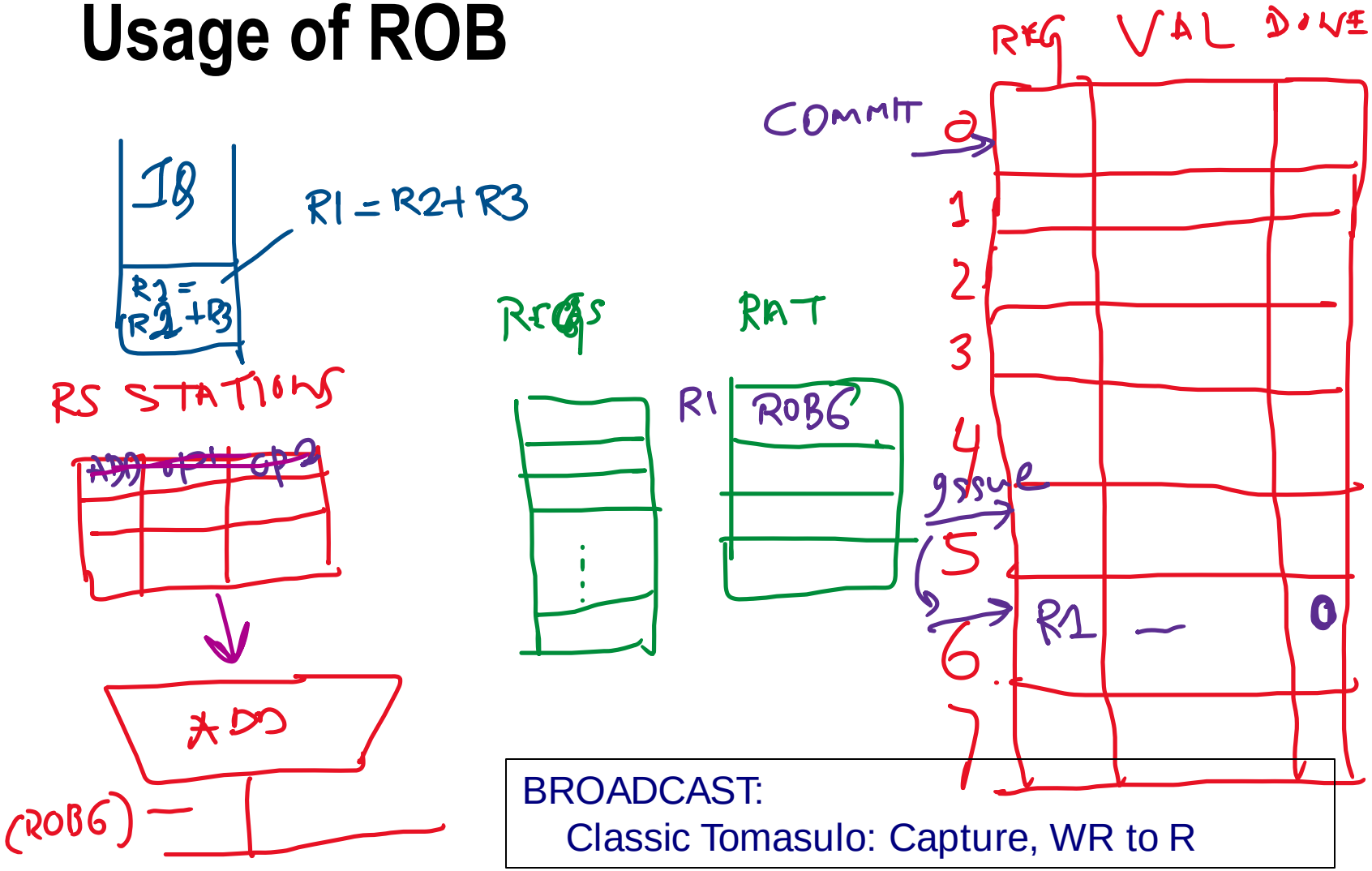
Usage of ROB



DISPATCH:
Ready? Send to Execute, Free RS

In classic Tomasulo, RS is emptied only after broadcast. This is because RS number is used in the RAT. Here the RAT points to a ROB address. So here RS can be freed during DISPATCH.

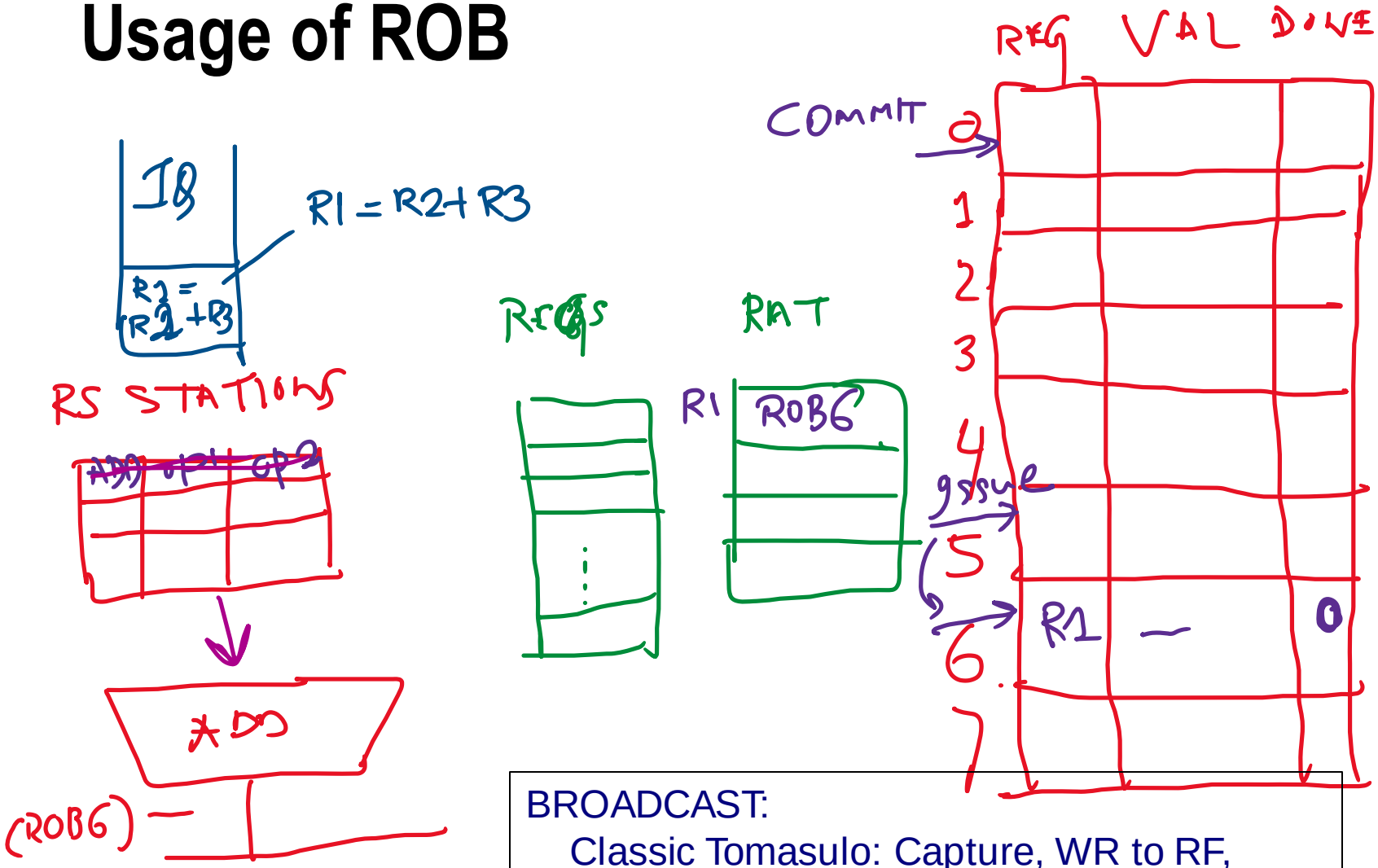
Usage of ROB



BROADCAST:
Classic Tomasulo: Capture, WR to R

In classic Tomasulo, RS is emptied only after broadcast. This is because RS number is used in the RAT. Here the RAT points to a ROB address. So here RS can be freed during DISPATCH.

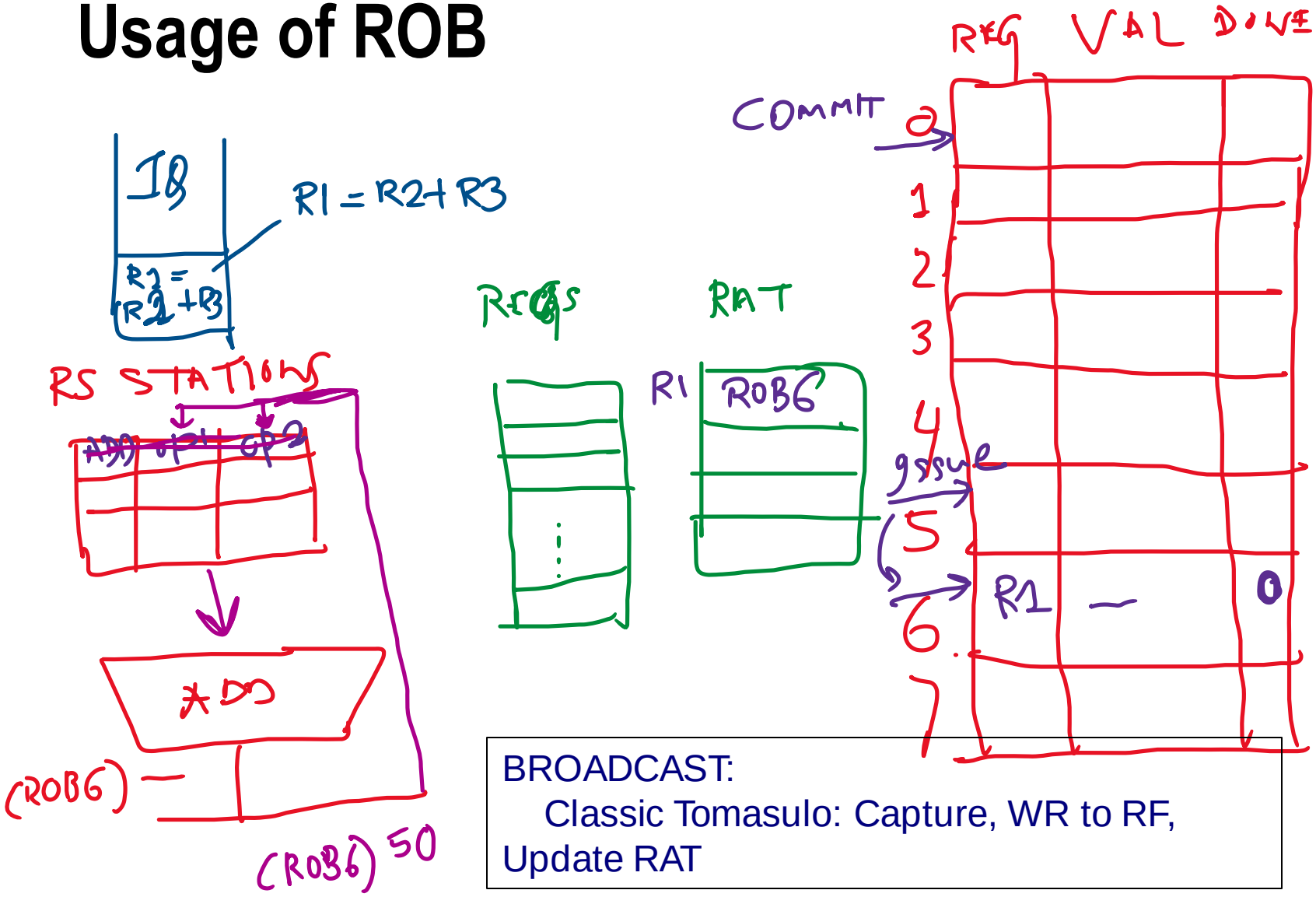
Usage of ROB



BROADCAST:

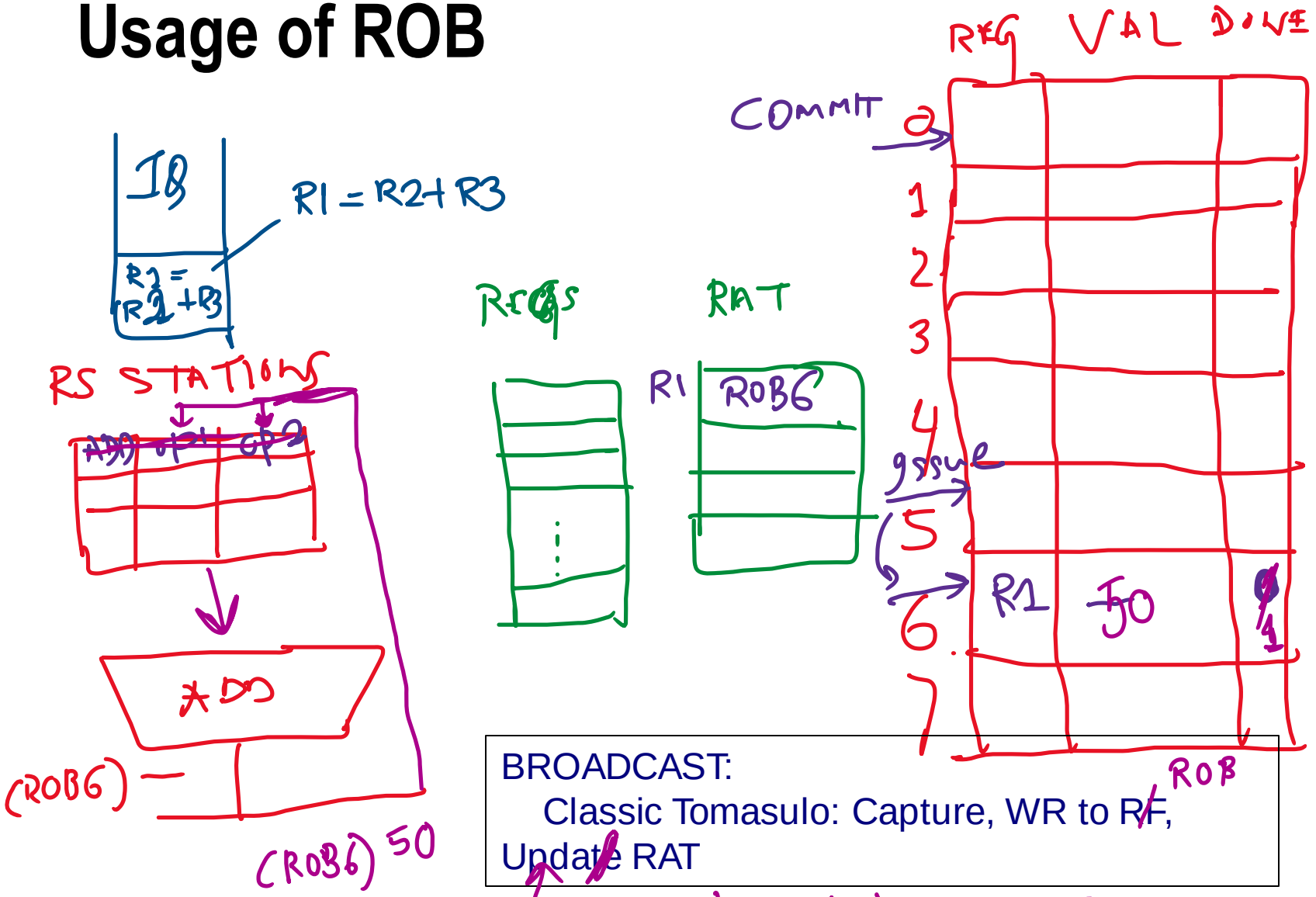
Classic Tomasulo: Capture, WR to RF, Update RAT

Usage of ROB



BROADCAST:
Classic Tomasulo: Capture, WR to RF, Update RAT

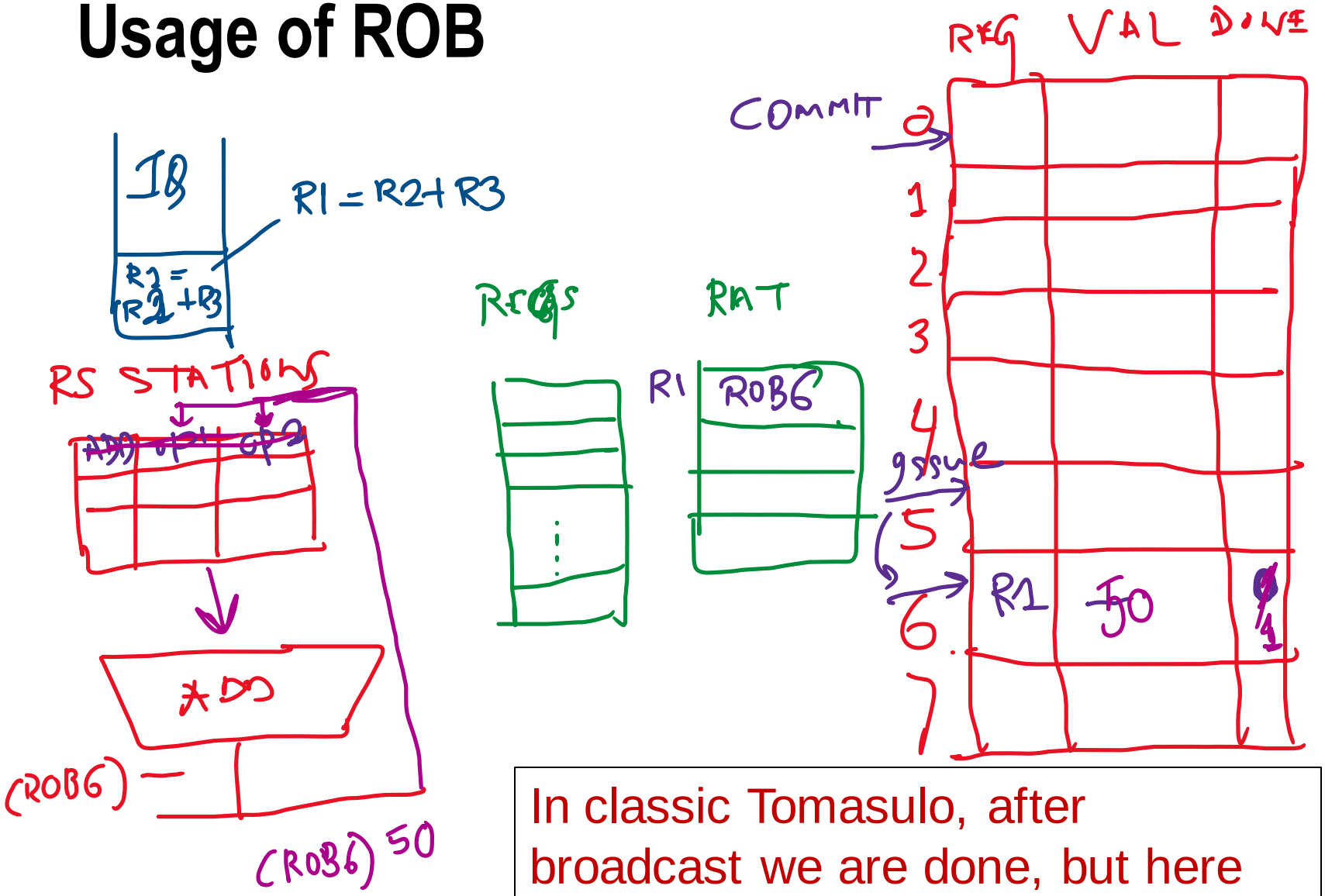
Usage of ROB



BROADCAST:
Classic Tomasulo: Capture, WR to RF, Update RAT

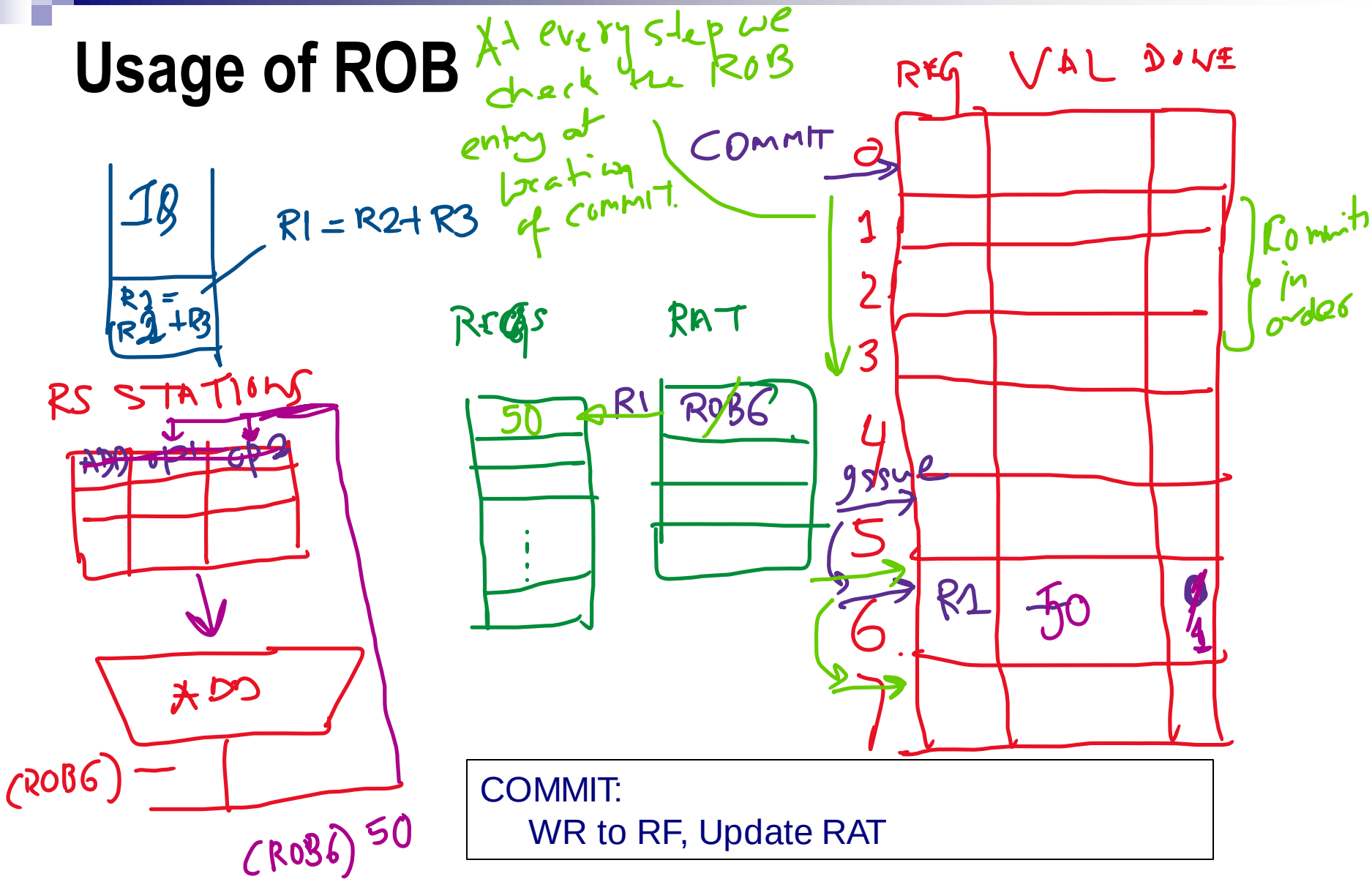
As we dont update Reg File, RAT is not updated.

Usage of ROB



In classic Tomasulo, after broadcast we are done, but here we have not yet written to the RF!

Usage of ROB



COMMIT:

WR to RF, Update RAT

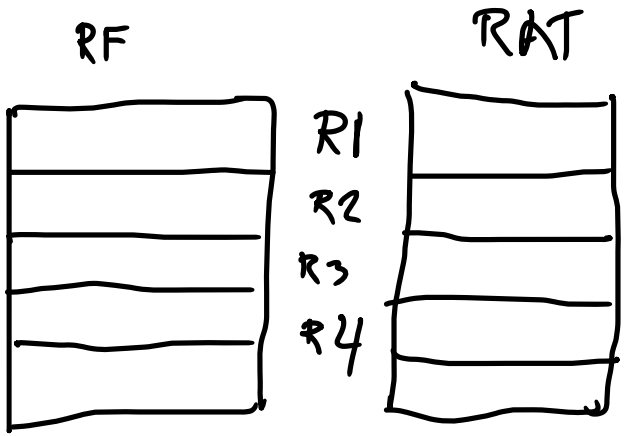
Key Differences with Classic Tomasulo

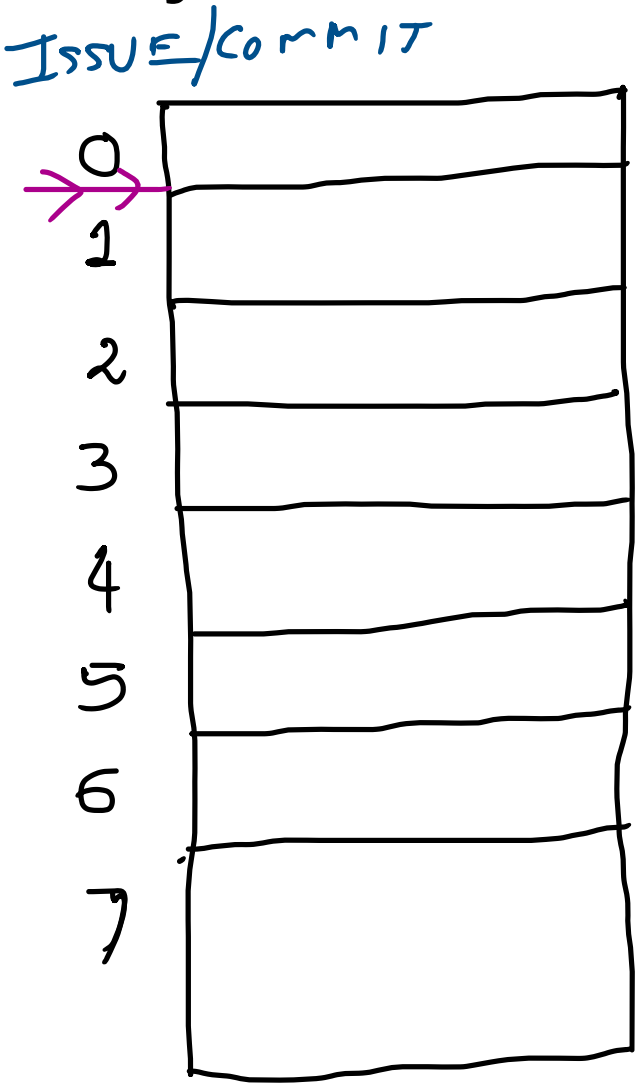
- Point RAT to ROB entry (not RS entry)
- Free RS on Dispatch (not Broadcast)
- Broadcast does not WR to RF, and does not update RAT
- New step, Commit for WR to RF and update RAT

Branch Misprediction Recovery

Instruction
LD R1, 0(R1)
BNE R1, R2, label
ADD R2, R1, R1
MUL R3, R3, R4
DIV R2, R3, R7

We predict these instructions are to be executed when actually not.

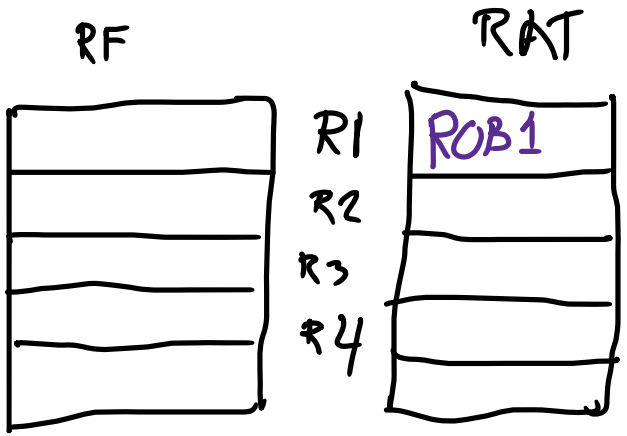


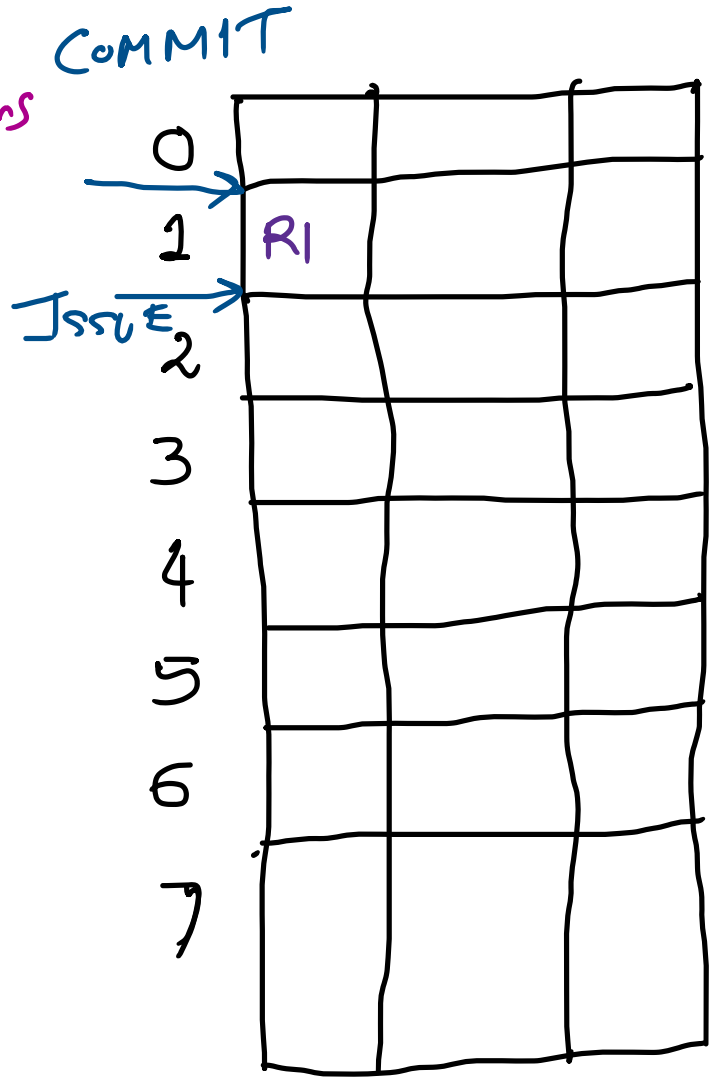


Branch Misprediction Recovery

Instruction
LD R1, 0(R1)
BNE R1, R2, label
ADD R2, R1, R1
MUL R3, R3, R4
DIV R2, R3, R7

We predict these instructions are to be executed when actually not.



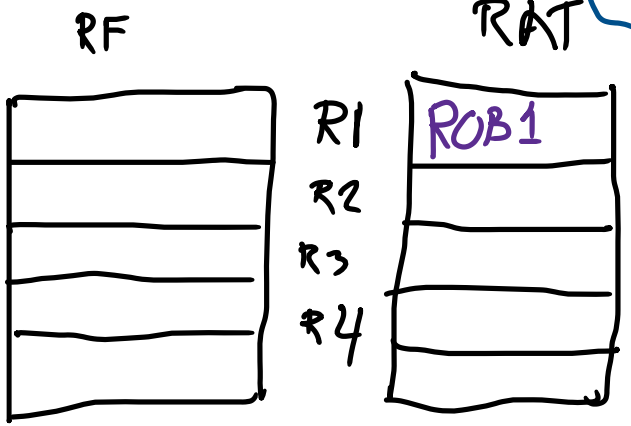


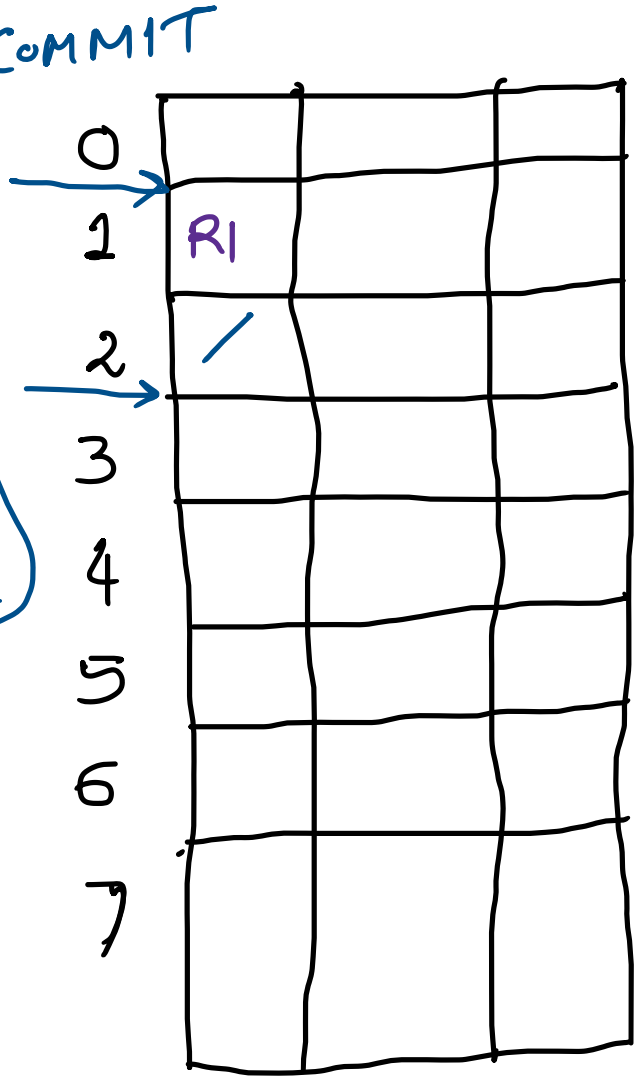
Branch Misprediction Recovery

Instruction
LD R1, 0(R1)
BNE R1, R2, label
ADD R2, R1, R1
MUL R3, R3, R4
DIV R2, R3, R7

We predict these instructions are to be executed when actually not.

No output, so we don't rename

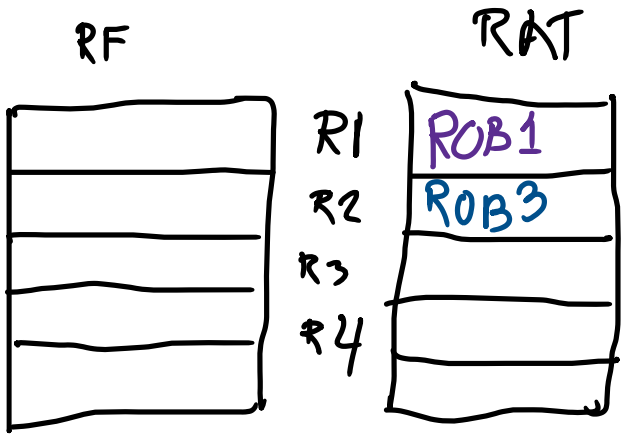


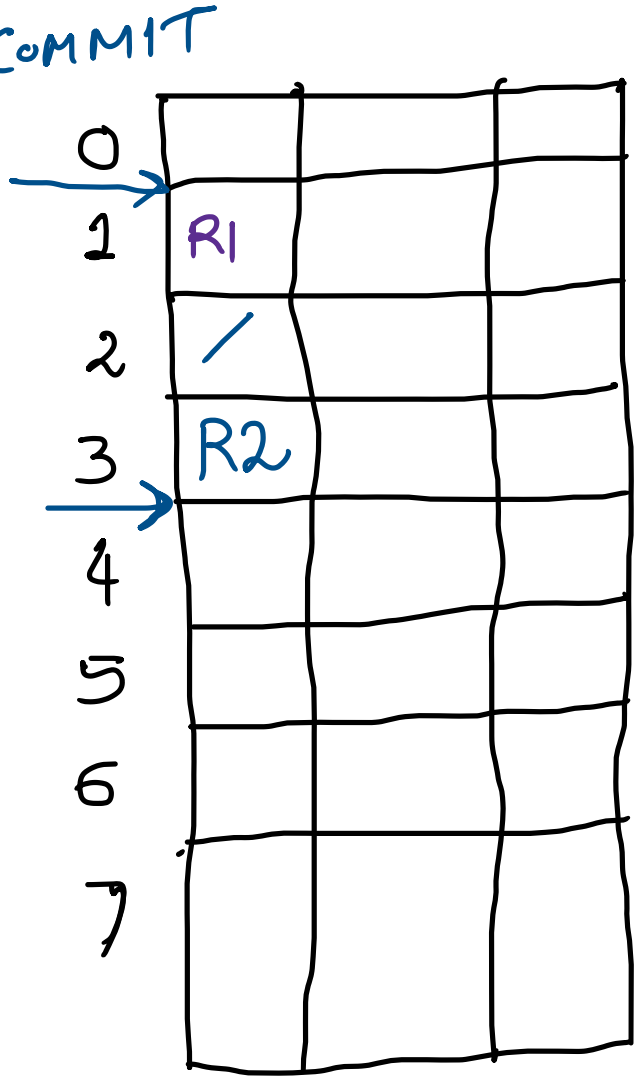


Branch Misprediction Recovery

Instruction
LD R1, 0(R1)
BNE R1, R2, label
ADD R2, R1, R1
MUL R3, R3, R4
DIV R2, R3, R7

We predict these instructions are to be executed when actually not.

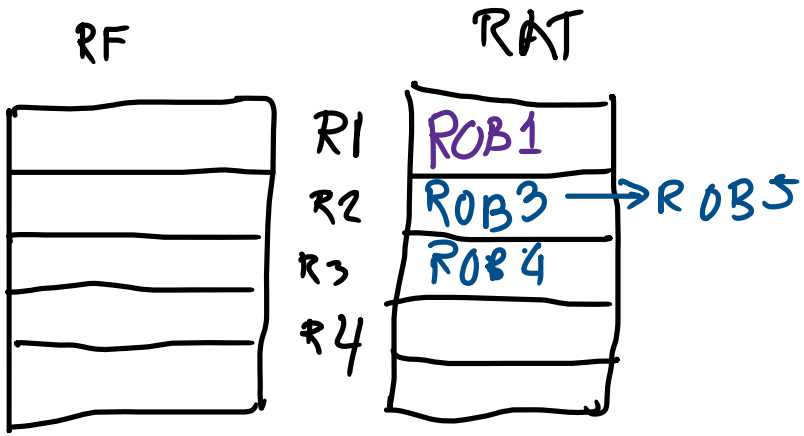


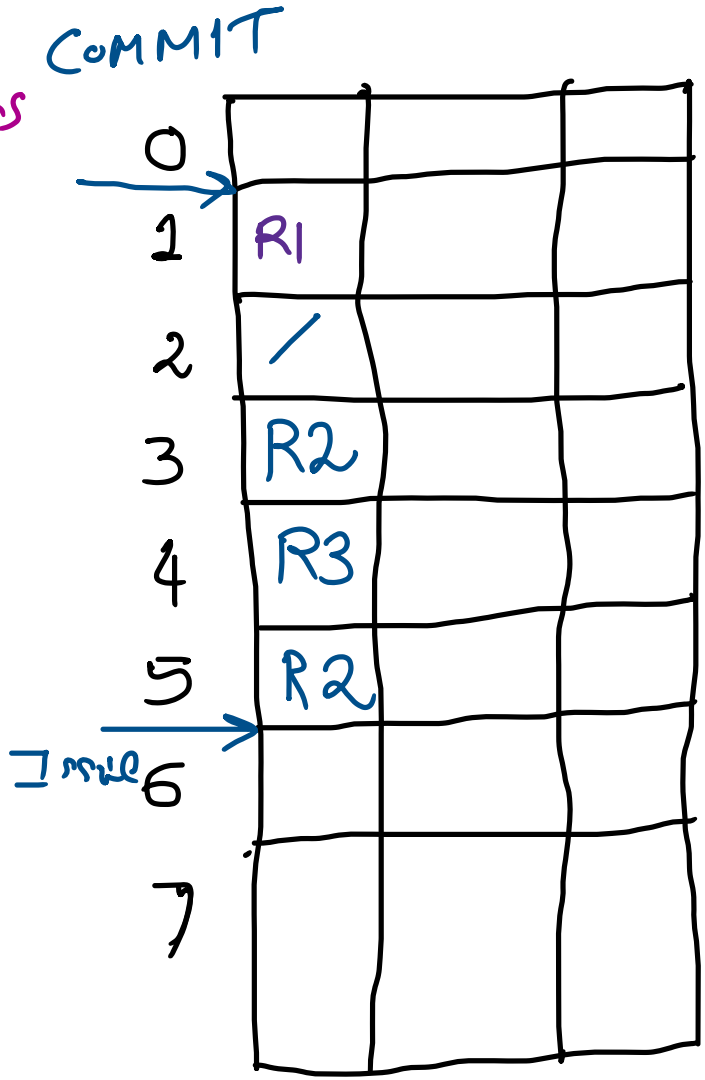


Branch Misprediction Recovery

Instruction
LD R1, 0(R1)
BNE R1, R2, label
ADD R2, R1, R1
MUL R3, R3, R4
DIV R2, R3, R7

We predict these instructions are to be executed when actually not.

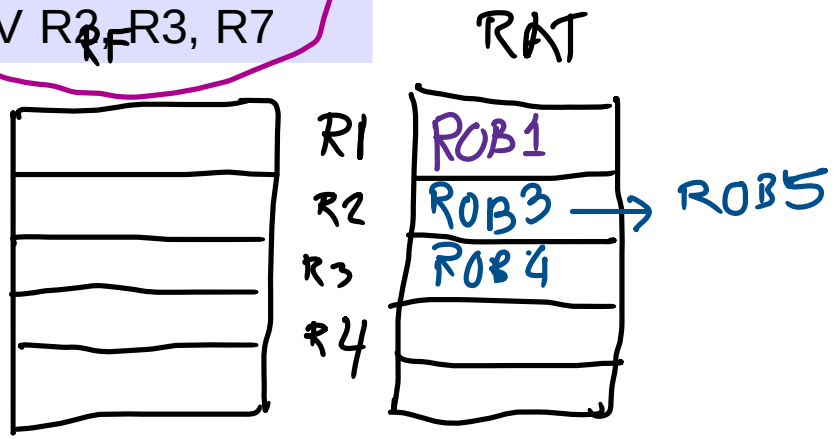


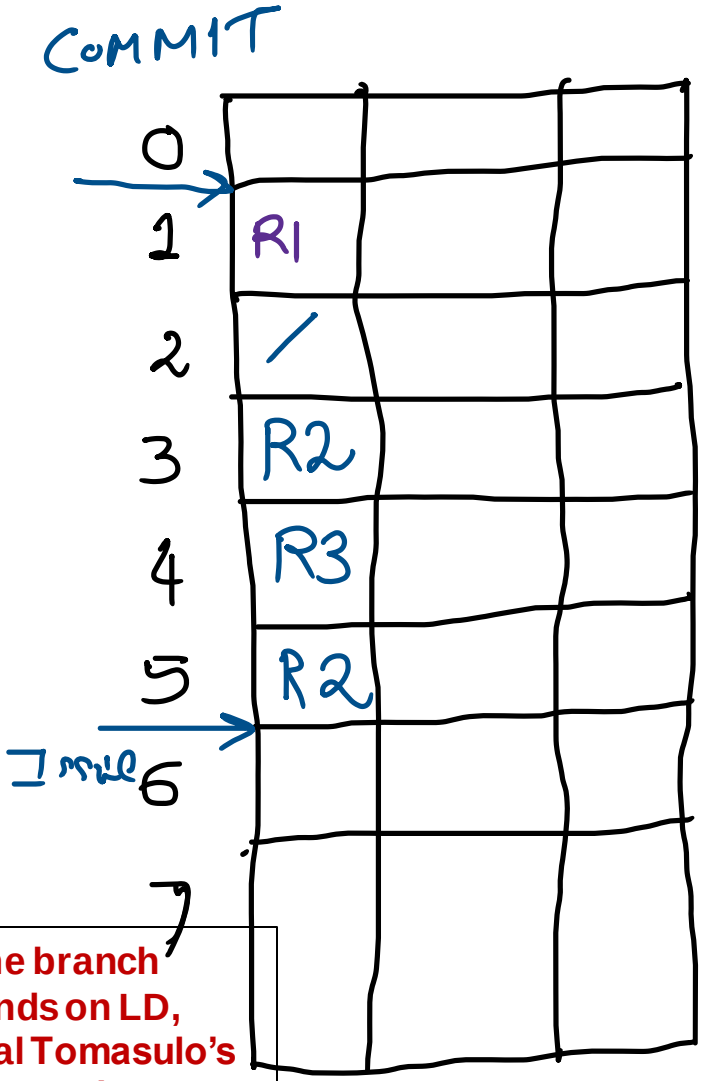


Branch Misprediction Recovery

Instruction
LD R1, 0(R1)
BNE R1, R2, label
ADD R2, R1, R1
MUL R3, R3, R4
DIV R3, R3, R7

We predict these instructions are to be executed when actually not.



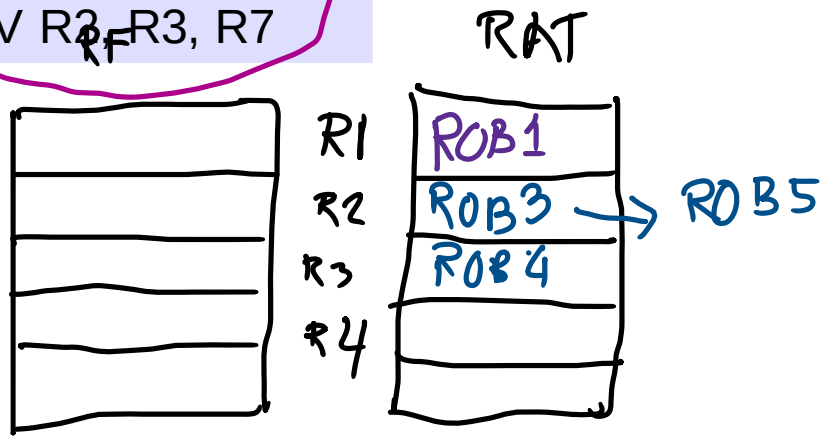


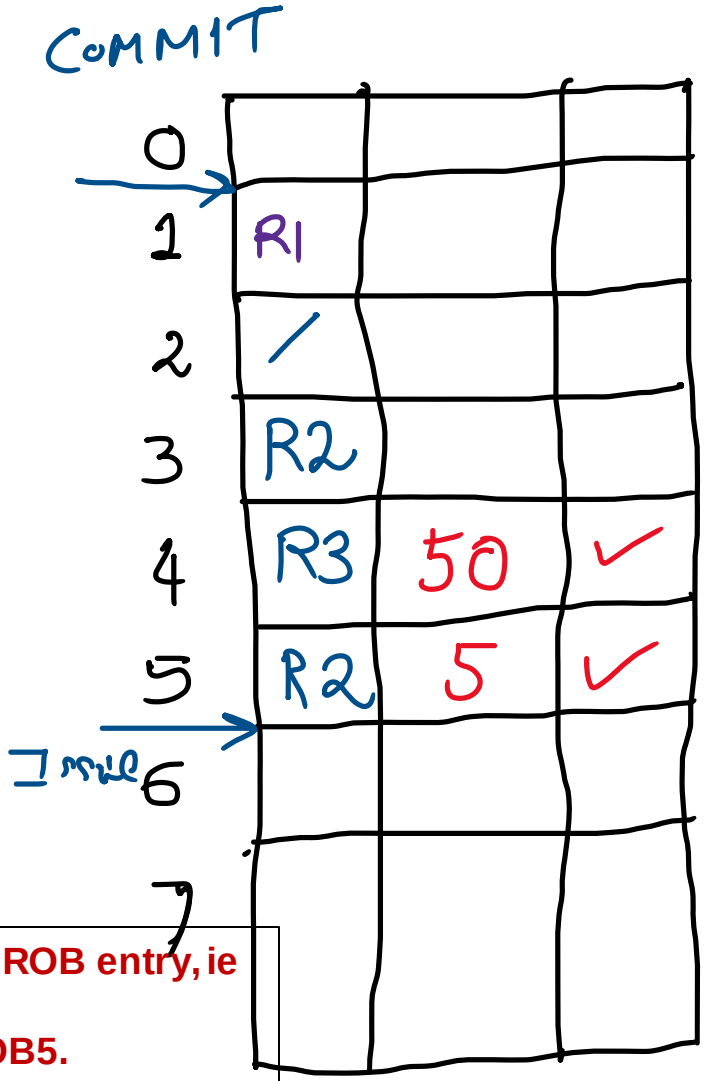
Suppose, the LD takes a long time because it is a cache miss. The branch cannot complete, because it depends on LD. The ADD also depends on LD, hence cannot complete. But the MUL can complete. So, in original Tomasulo's algorithm the result would be written to the register file and we do not know how to recover once we know that the branch is mis-predicted.

Branch Misprediction Recovery

Instruction
LD R1, 0(R1)
BNE R1, R2, label
ADD R2, R1, R1
MUL R3, R3, R4
DIV R3, R3, R7

We predict these instructions are to be executed when actually not.



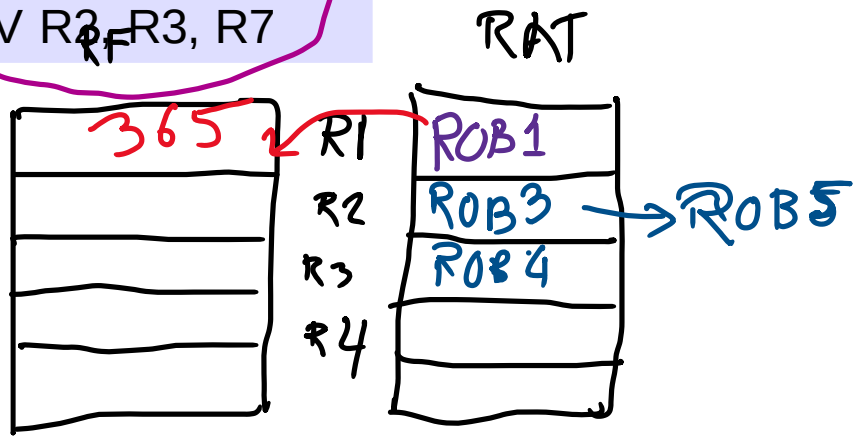


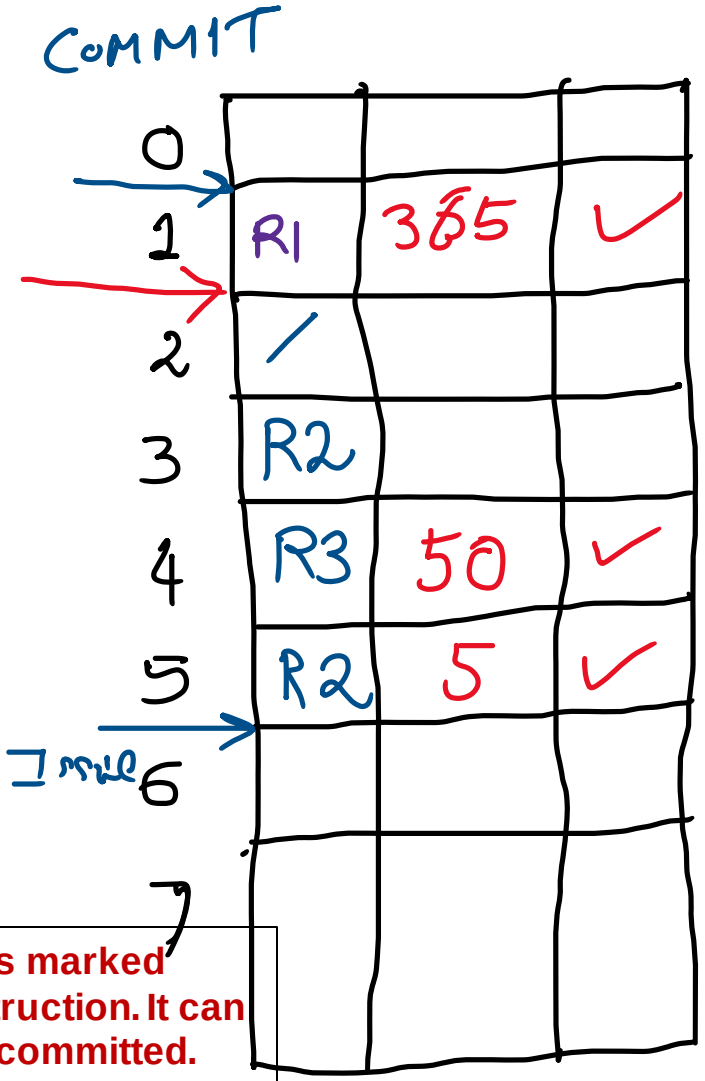
In our ROB based processor, the MUL result (say 50) goes to the ROB entry, ie here it gets written in ROB4.
Then the DIV can also begin, and its value (say 5) is written to ROB5.
The DONE bits are made 1 in the ROB.
Thus, here we just update the ROB entries, like a scratch-pad, while the registers are still unmodified.

Branch Misprediction Recovery

Instruction
LD R1, 0(R1)
BNE R1, R2, label
ADD R2, R1, R1
MUL R3, R3, R4
DIV R3, R3, R7

We predict these instructions are to be executed when actually not.



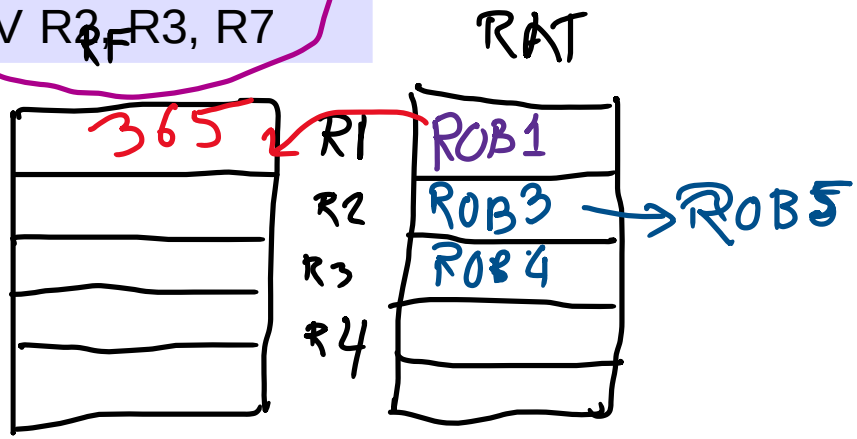


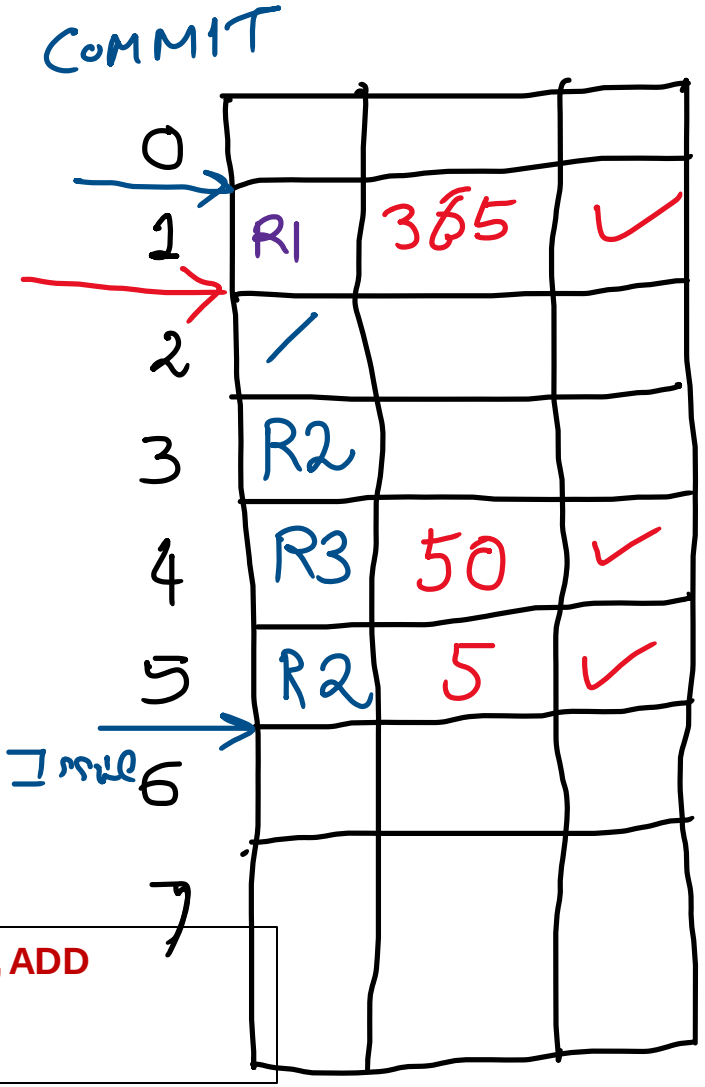
Eventually, the load is done and say it brings a value of 365 and is marked done. We are always checking if commit can be done on this instruction. It can be as it is marked done, and all previous instructions have been committed. We update the COMMIT pointer and write 365 in reg file.

Branch Misprediction Recovery

Instruction
LD R1, 0(R1)
BNE R1, R2, label
ADD R2, R1, R1
MUL R3, R3, R4
DIV R3, R3, R7

We predict these instructions are to be executed when actually not.



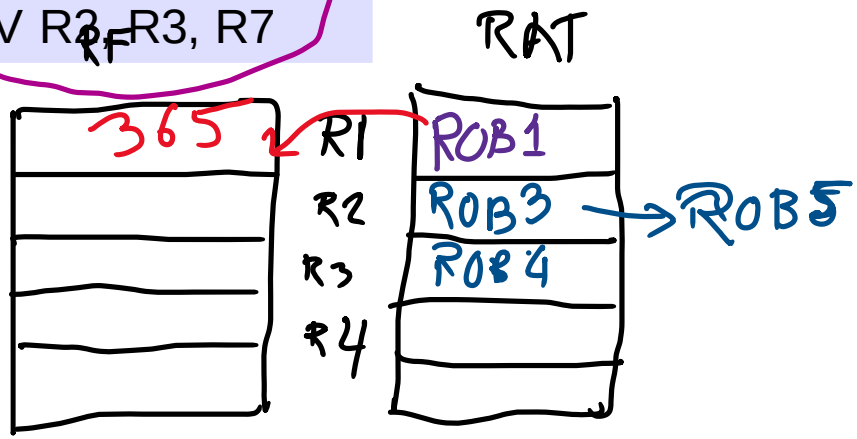


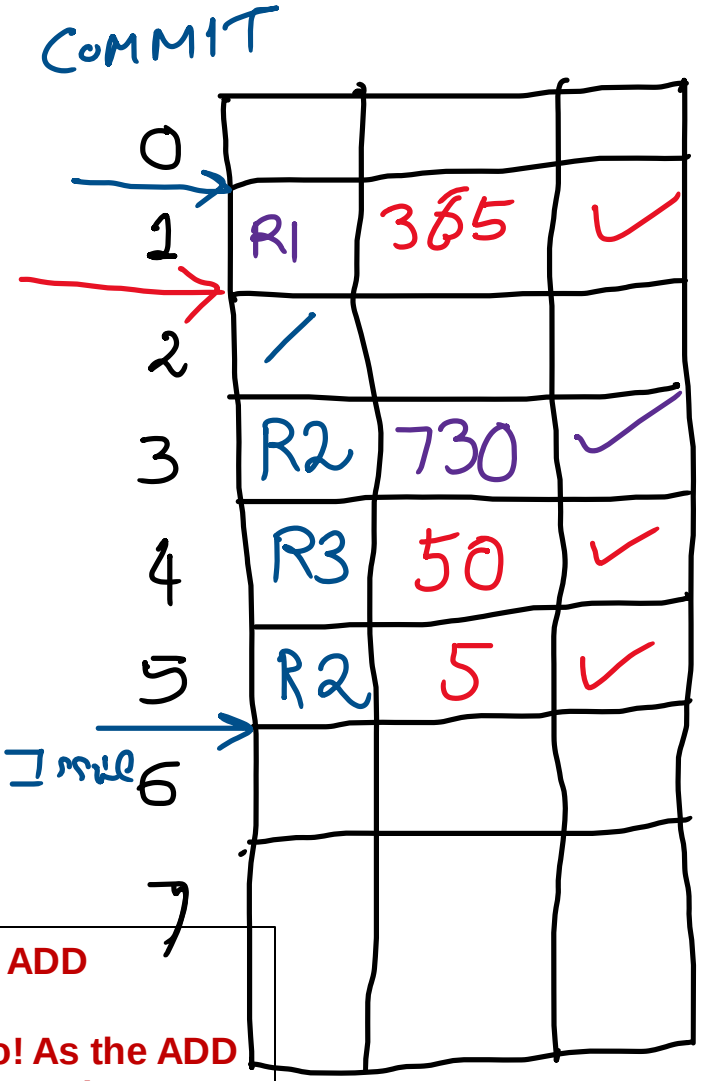
Lets assume that the Branch takes longer time than the ADD. So, ADD completes and produces a value, say 730.
In Tomasulo's algorithm would we have written this to the RF?

Branch Misprediction Recovery

Instruction
LD R1, 0(R1)
BNE R1, R2, label
ADD R2, R1, R1
MUL R3, R3, R4
DIV R3, R3, R7

We predict these instructions are to be executed when actually not.



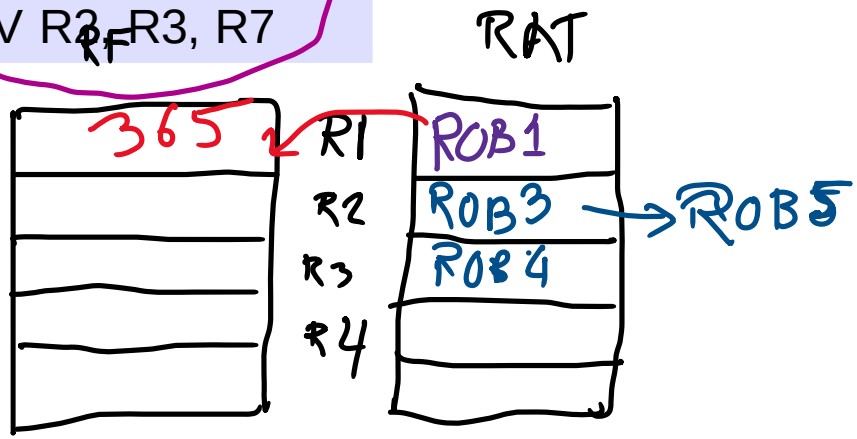


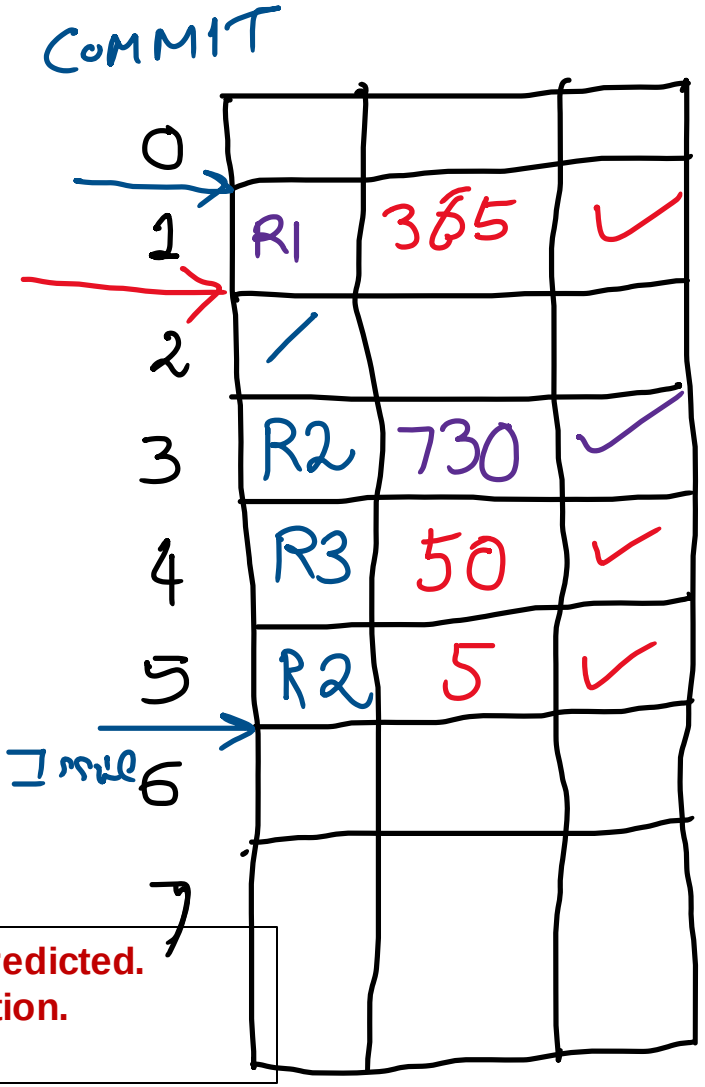
Lets assume that the Branch takes longer time than the ADD. So, ADD completes and produces a value, say 730.
In Tomasulo's algorithm would we have written this to the RF? No! As the ADD is not the most recent instruction writing to R2, we would not have written to the RF also there. In ROB based processor, we of course just write to the ROB.

Branch Misprediction Recovery

Instruction
LD R1, 0(R1)
BNE R1, R2, label
ADD R2, R1, R1
MUL R3, R3, R4
DIV R3, R3, R7

We predict these instructions are to be executed when actually not.



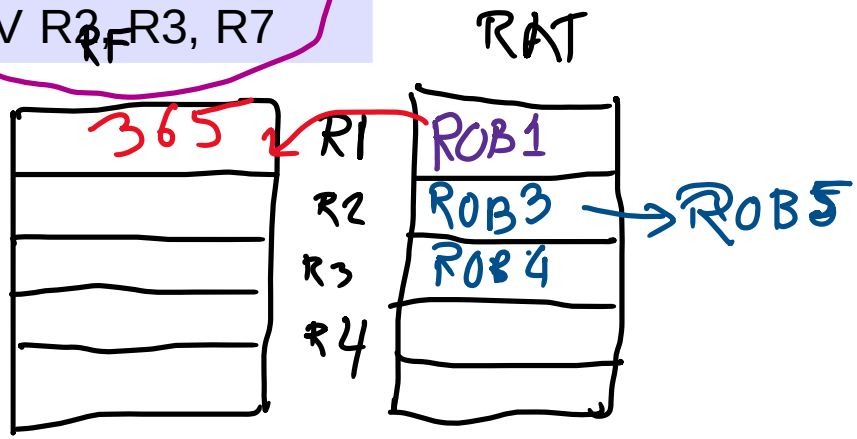


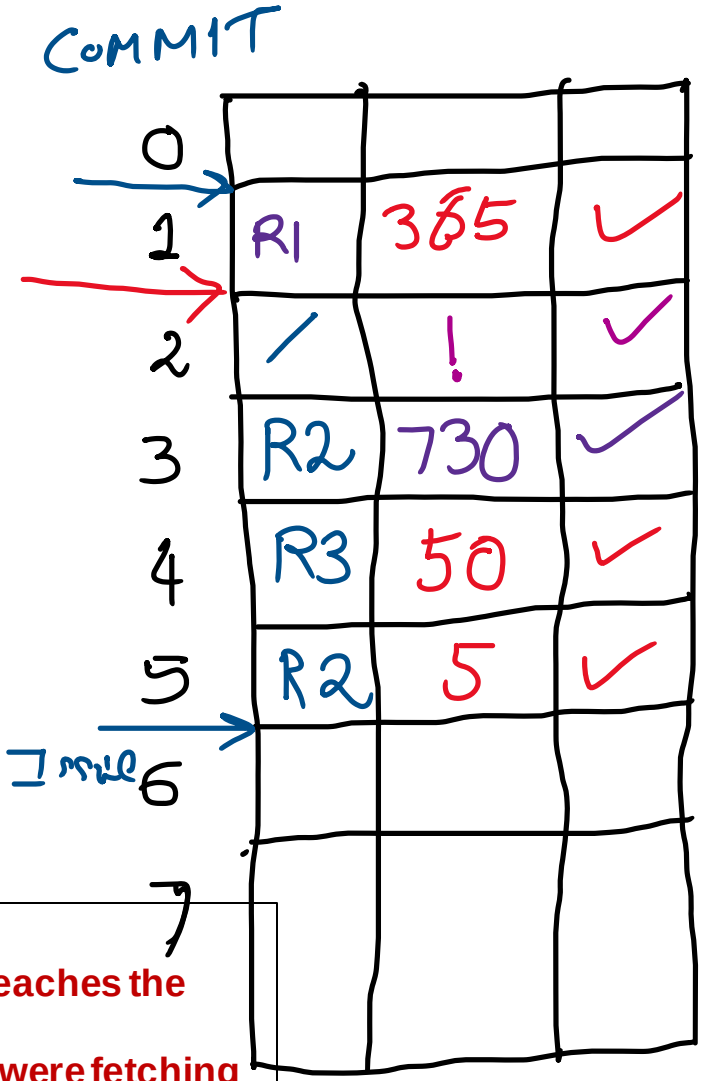
Eventually, the branch is resolved. We figure that we have mis-predicted.
So we mark done and can start fetching from the correct instruction.
But how do we undo the wrong instructions?

Branch Misprediction Recovery

Instruction
LD R1, 0(R1)
BNE R1, R2, label
ADD R2, R1, R1
MUL R3, R3, R4
DIV R3, R3, R7

We predict these instructions are to be executed when actually not.



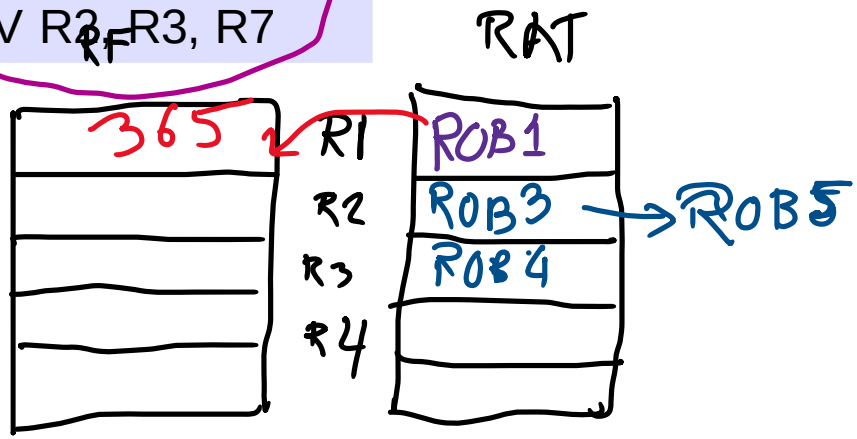


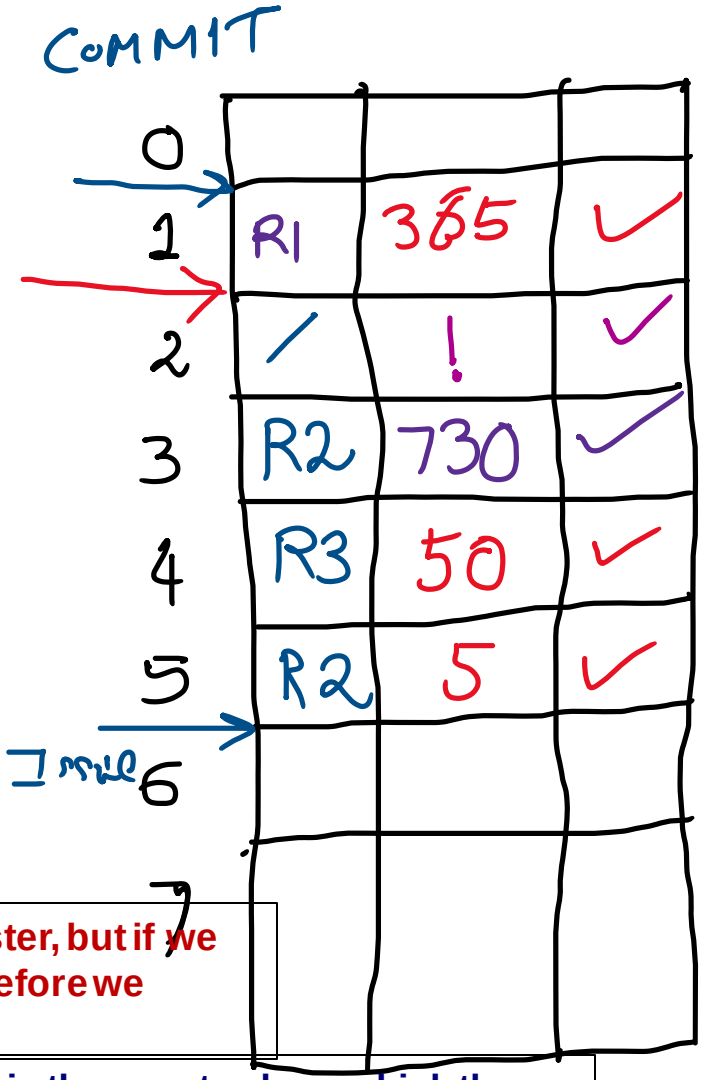
We annotate a '!' in ROB2 to indicate the branch outcome.
We continue executing the wrong instructions until the commit reaches the branch instruction.
When we commit the branch, we realize that after the branch we were fetching wrong instructions. The PC that the branch should have updated by the branch is different from the one used.

Branch Misprediction Recovery

Instruction
LD R1, 0(R1)
BNE R1, R2, label
ADD R2, R1, R1
MUL R3, R3, R4
DIV R3, R3, R7

We predict these instructions are to be executed when actually not.



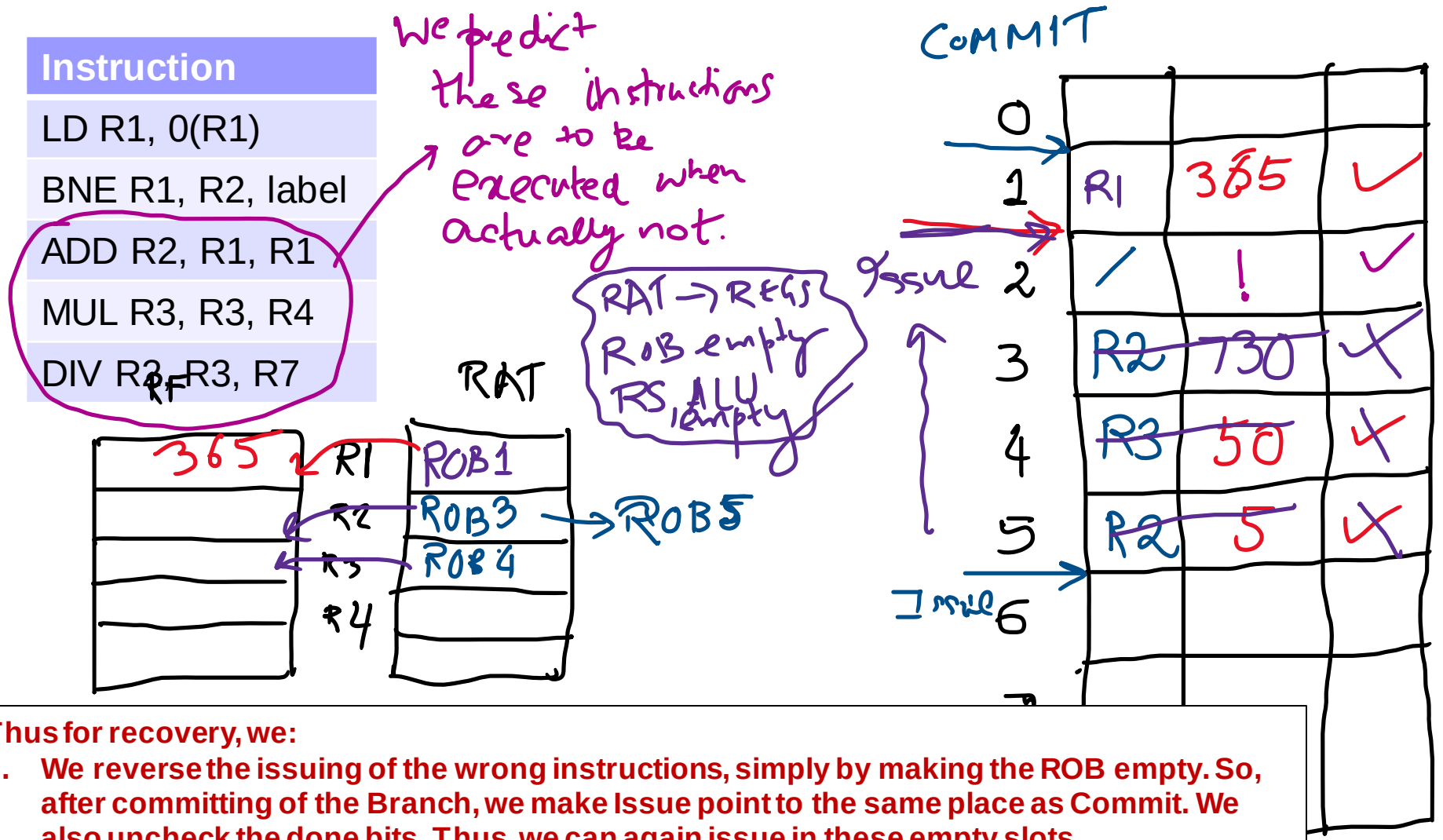


At that point we commit the branch. It does not write to any register, but if we see a '!' (basically a symbol for mis-prediction) we do recovery before we restart or fetch from the next correct PC.

Note, at the point of the commit of the branch, the registers contain the exact values which they should contain. All prior instructions have been committed, which means their updates have been reflected in the RF. Further none of the wrong instructions have updated the RF.

Branch Misprediction Recovery

Instruction
LD R1, 0(R1)
BNE R1, R2, label
ADD R2, R1, R1
MUL R3, R3, R4
DIV R3, R3, R7



Thus for recovery, we:

1. **We reverse the issuing of the wrong instructions, simply by making the ROB empty. So, after committing of the Branch, we make Issue point to the same place as Commit. We also uncheck the done bits. Thus, we can again issue in these empty slots.**
2. **Also the RAT now contains wrong values. Note the RF contains correct data. So for undoing the damage, we just make the RAT to point to corresponding registers.**

ROB & Exceptions

Instructions
DIV R0,R1,R2
ADD R1,…

Can be delayed: It can realize R2=0 much later.

The ADD is much faster!

In Tomasulo, it will write to R1 much before R2==0 is detected & that we should jump to exception handler & not execute the ADD

ROB & Exceptions

Instructions
DIV R0,R1,R2
ADD R1,…

In ROB based processors, we treat the exception just as another result. Thus instead of a conventional result of the DIV, we mark ‘EXC’ (or some symbol) in the ROB, for R0.

When the DIV reaches commit, the ADD has still not committed, and everything before DIV did.
So, we simply flush everything including DIV.
Then jump to the exception handler.
Thus,

- Everything before DIV is committed.
- Everything after that including DIV is not committed.
- The exception handler should be seeing exactly this stable state.

Page Fault Exception

Instructions
LD R0,0(R2)
ADD R2,,…

When the commit reaches the page fault, we have committed everything before the page fault, but have not committed the load itself and anything after.

So, again this is a good stable point for the page fault exception handler.
When we load the page from the disk and continue executing the load, it continues seamlessly as nothing (no wrong instruction) really executed!

Phantom Exceptions

Instructions
BEQ R1,R2,label
DIV R0, R0, R5

If we predict branch is not taken, then we execute the DIV and may get an exception.
By that time, the branch is not resolved!
When we do resolve, it may be late, because the Div-by-0 has already triggered? How does the ROB handle this?

Phantom Exceptions

Instructions
BEQ R1,R2,label
DIV R0, R0, R5

If we predict branch is not taken, then we execute the DIV and may get an exception. By that time, the branch is not resolved! When we do resolve, it may be late, because the Div-by-0 has already triggered? How does the ROB handle this?

The result of the exception is marked as 'EXC' in the ROB.
As the Commit pointer reaches the branch, we determine '!'-misprediction.
So, we cancel the following instructions in the ROB (as before), as DIV onwards (anything after the branch) nothing has committed.

(1) Write Exception as a result in the ROB

(2) Delay actual handling of exception until commit.

Exception with ROB Quiz

Instructions	Status	New Status
ADD R2,R2,R1	COMMITTED	
LW R1,0(R2)	EXECUTING	
ADD R3,R4,R5	DONE	
DIV R3,R2,R3	EXECUTING EXCEPTION	
ADD R1,R4,R4	DONE	
ADD R3,R2,R2	DONE	

What is the new status of the instructions at which the exception handler can be called?

Exception with ROB Quiz

Instructions	Status	New Status
ADD R2,R2,R1	COMMITTED	COMMITTED
LW R1,0(R2)	EXECUTING	COMMITTED
ADD R3,R4,R5	DONE	COMMITTED
DIV R3,R2,R3	EXECUTING EXCEPTION	
ADD R1,R4,R4	DONE	
ADD R3,R2,R2	DONE	

Exception with ROB Quiz

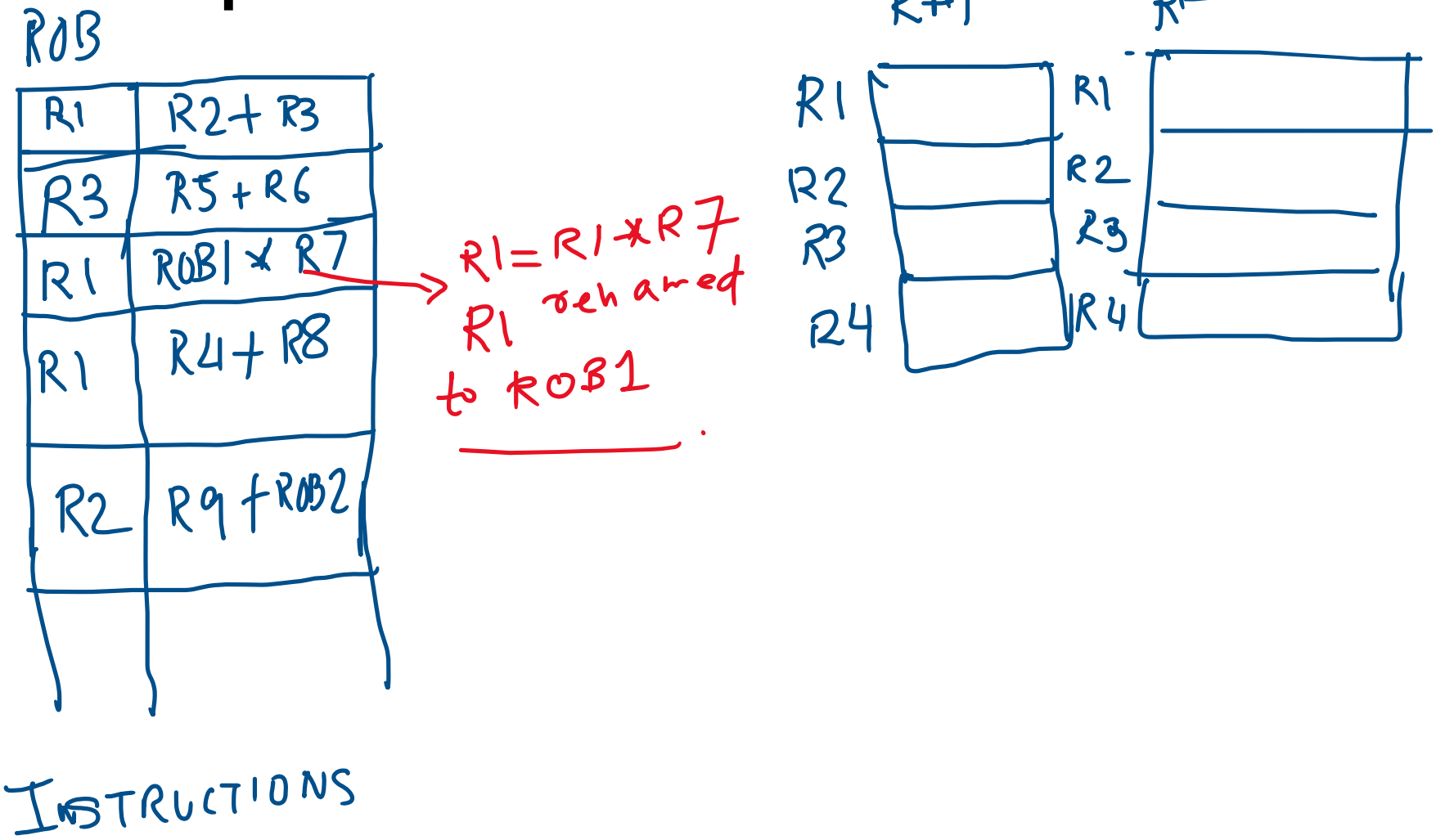
Instructions	Status	New Status
ADD R2,R2,R1	COMMITTED	COMMITTED
LW R1,0(R2)	EXECUTING	COMMITTED
ADD R3,R4,R5	DONE	COMMITTED
DIV R3,R2,R3	EXECUTING	INVALIDATE
ADD R1,R4,R4	DONE	INVALID
ADD R3,R2,R2	DONE	INVALID.

Commit/Issue → EXCEPTION

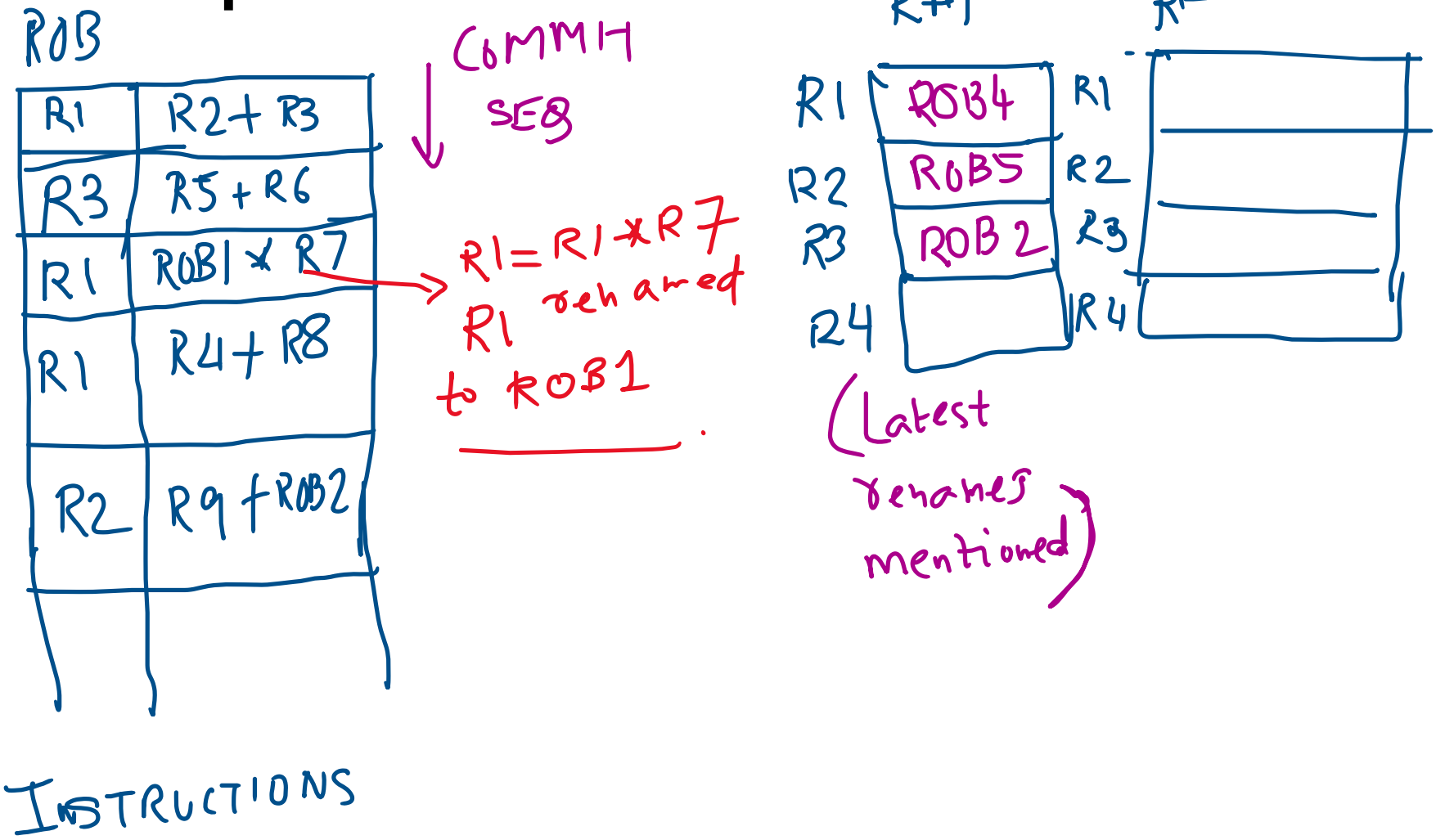
RAT Updates on Commit

- In the original Tomasulo's algoritm, the write to RF took place when the RAT mentions the same RS label as the current instruction intending to write (during broadcast)
- However, with ROB, we will always write to the RF.
- This is because at any time if there is an exception, etc. we just clear the ROB and the RAT.
- The RF contains the correct register contents till the latest committed instruction.

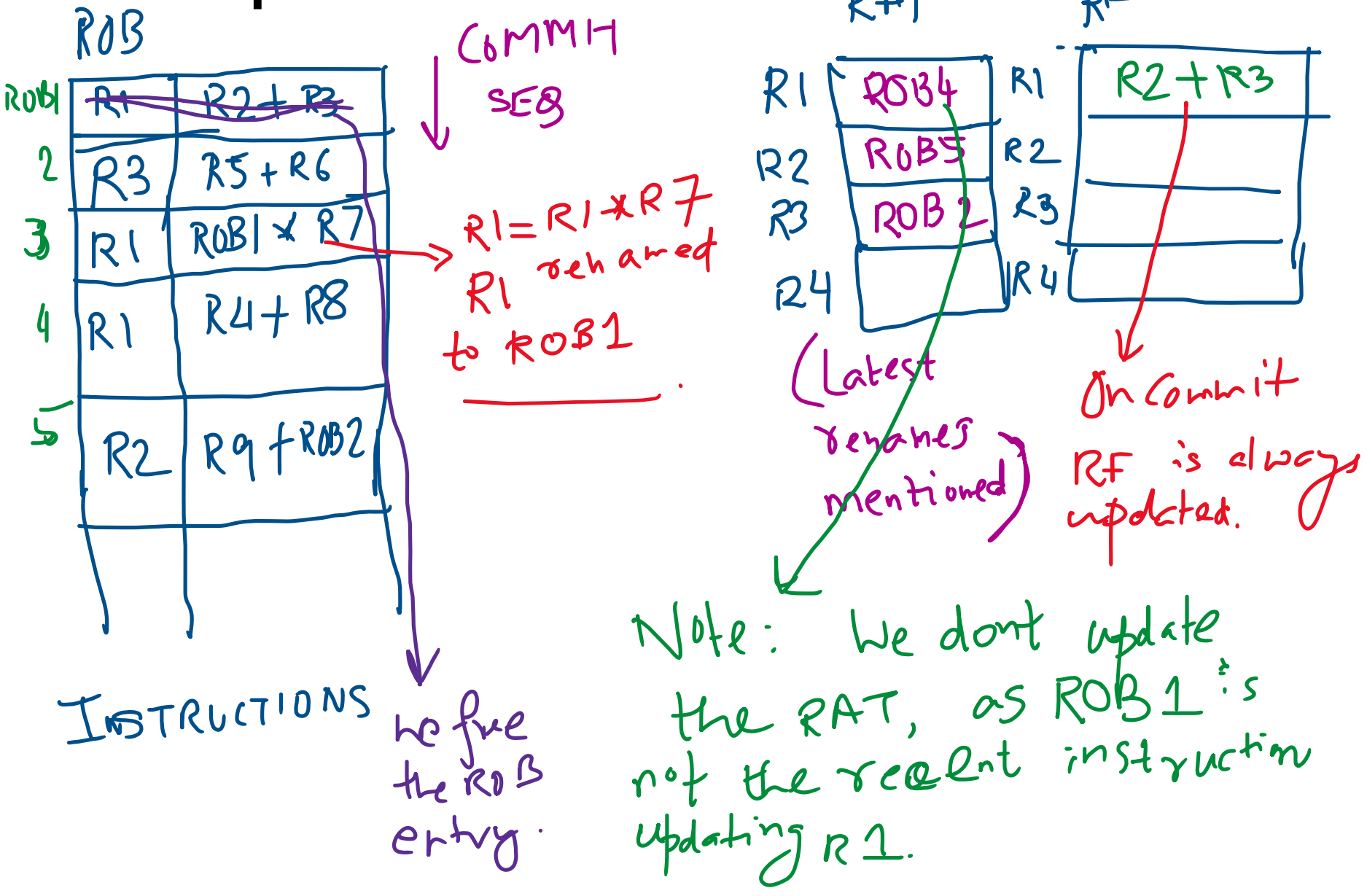
RAT Updates on Commit



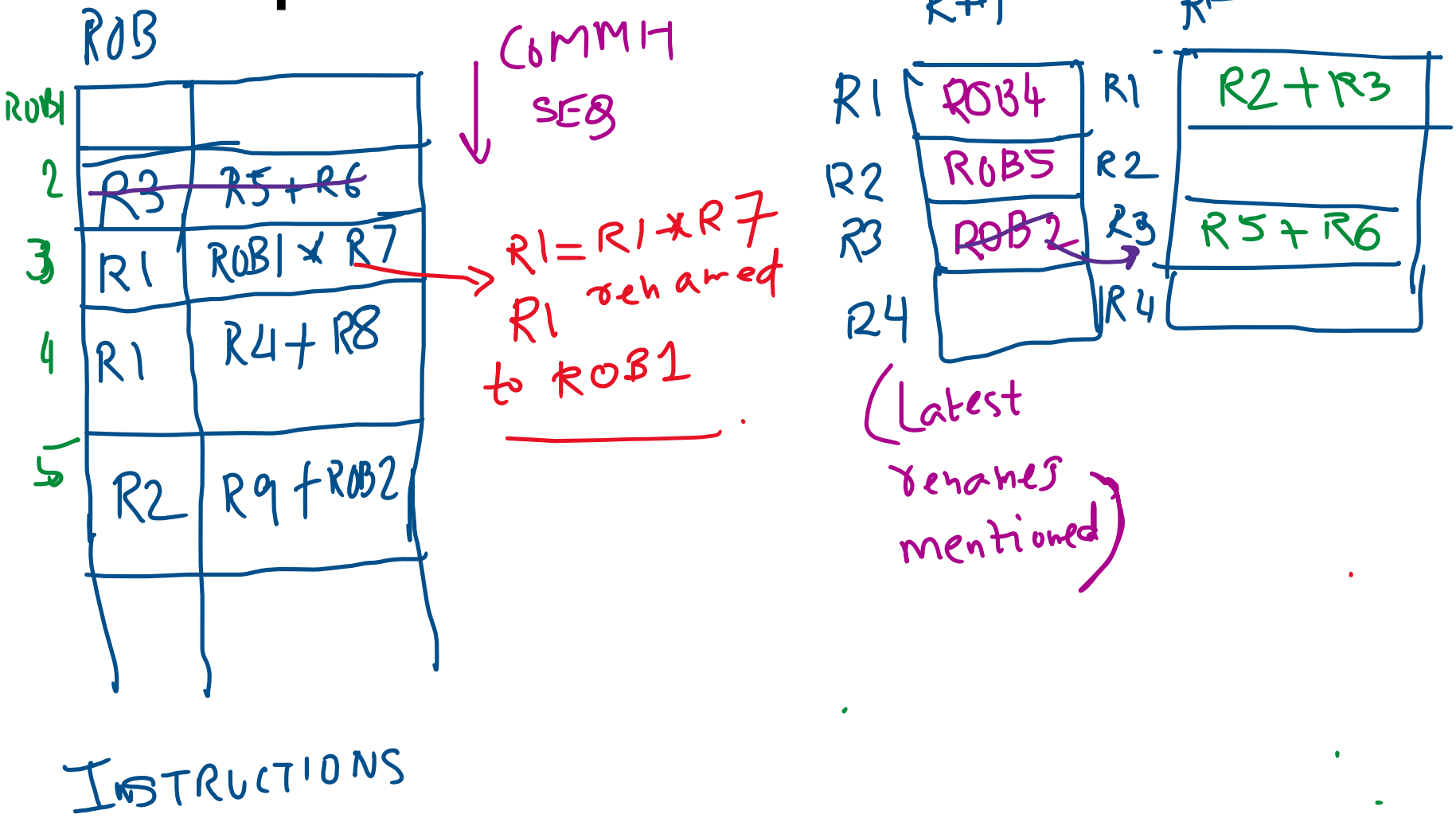
RAT Updates on Commit



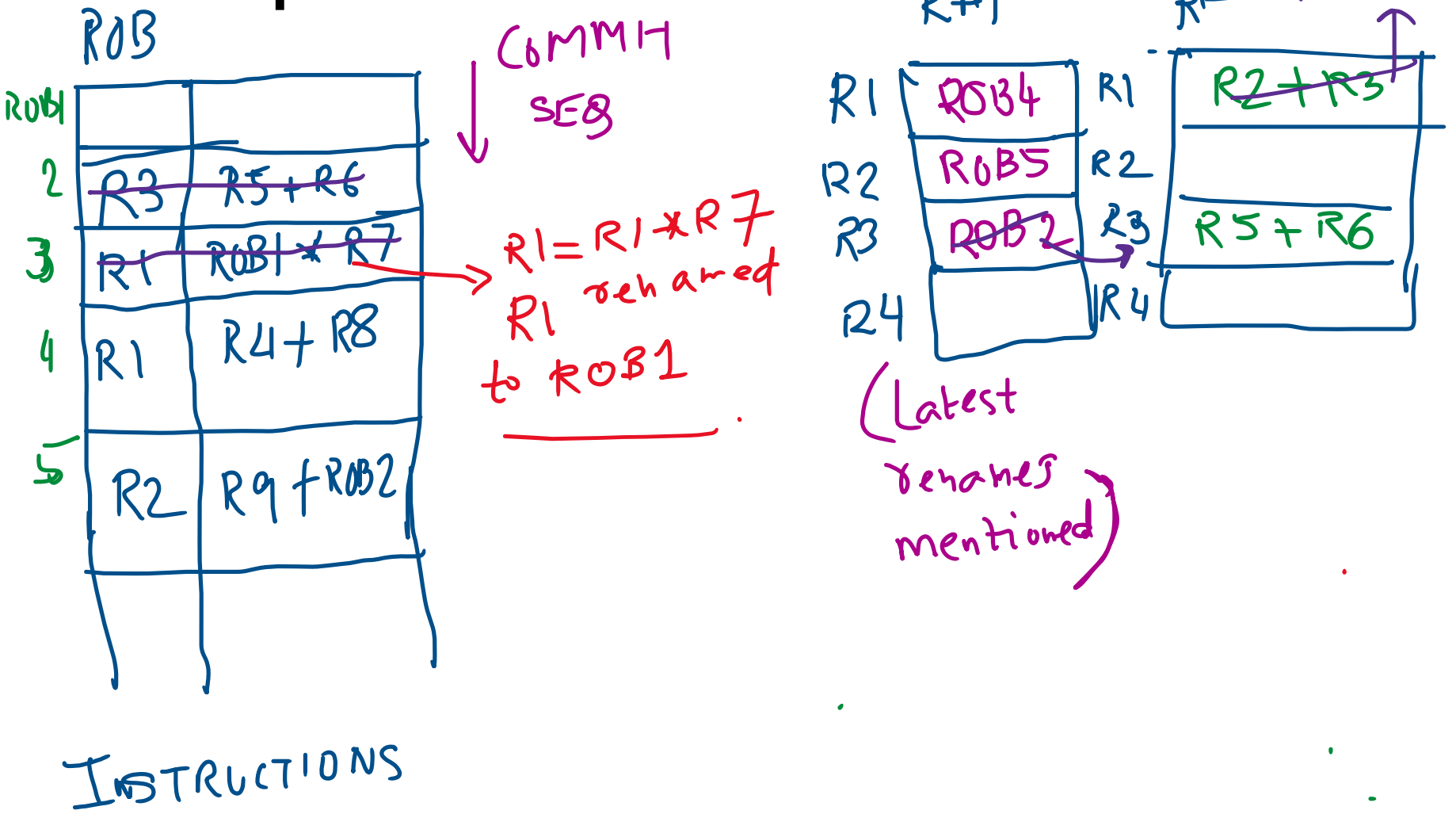
RAT Updates on Commit



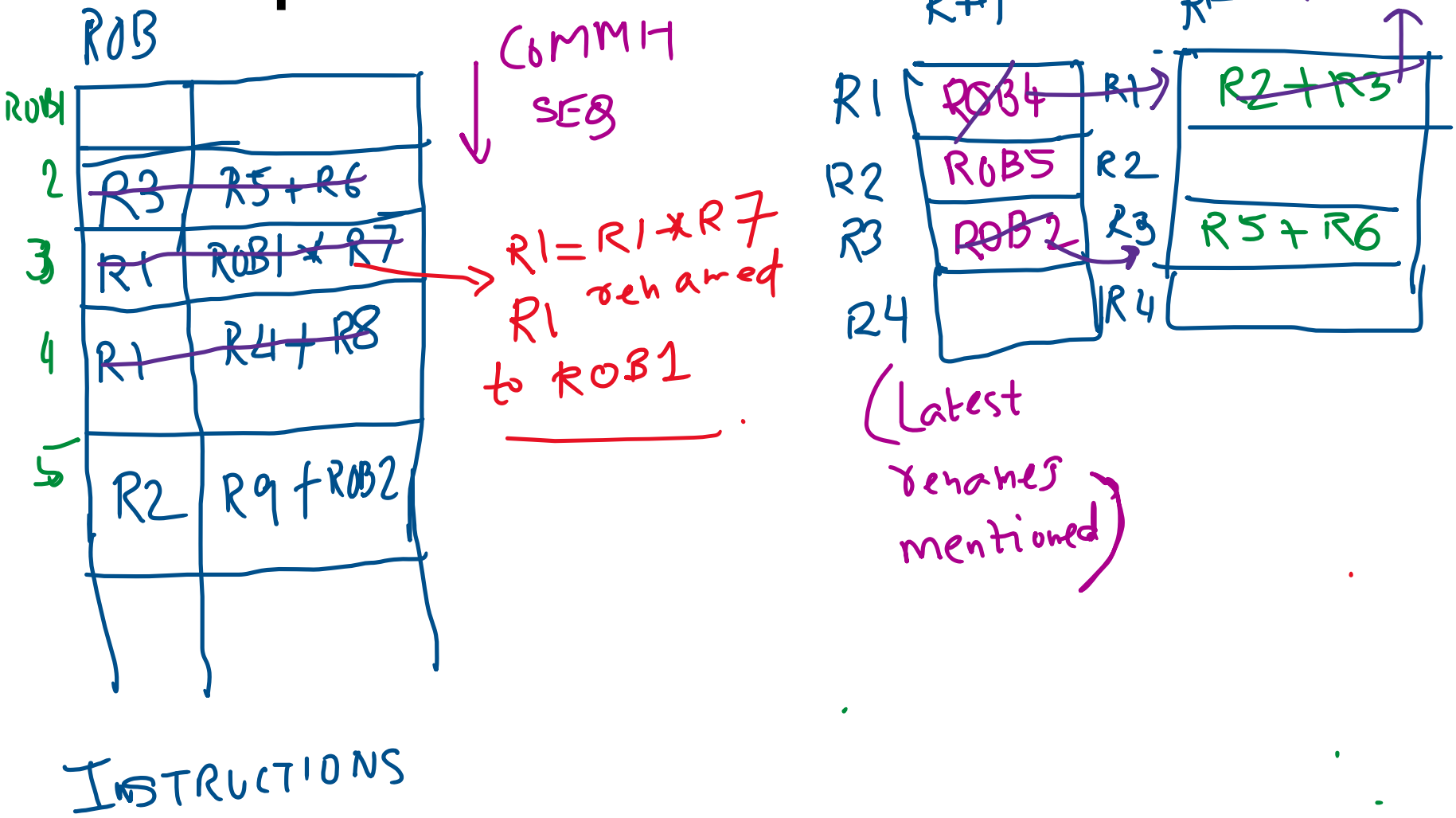
RAT Updates on Commit



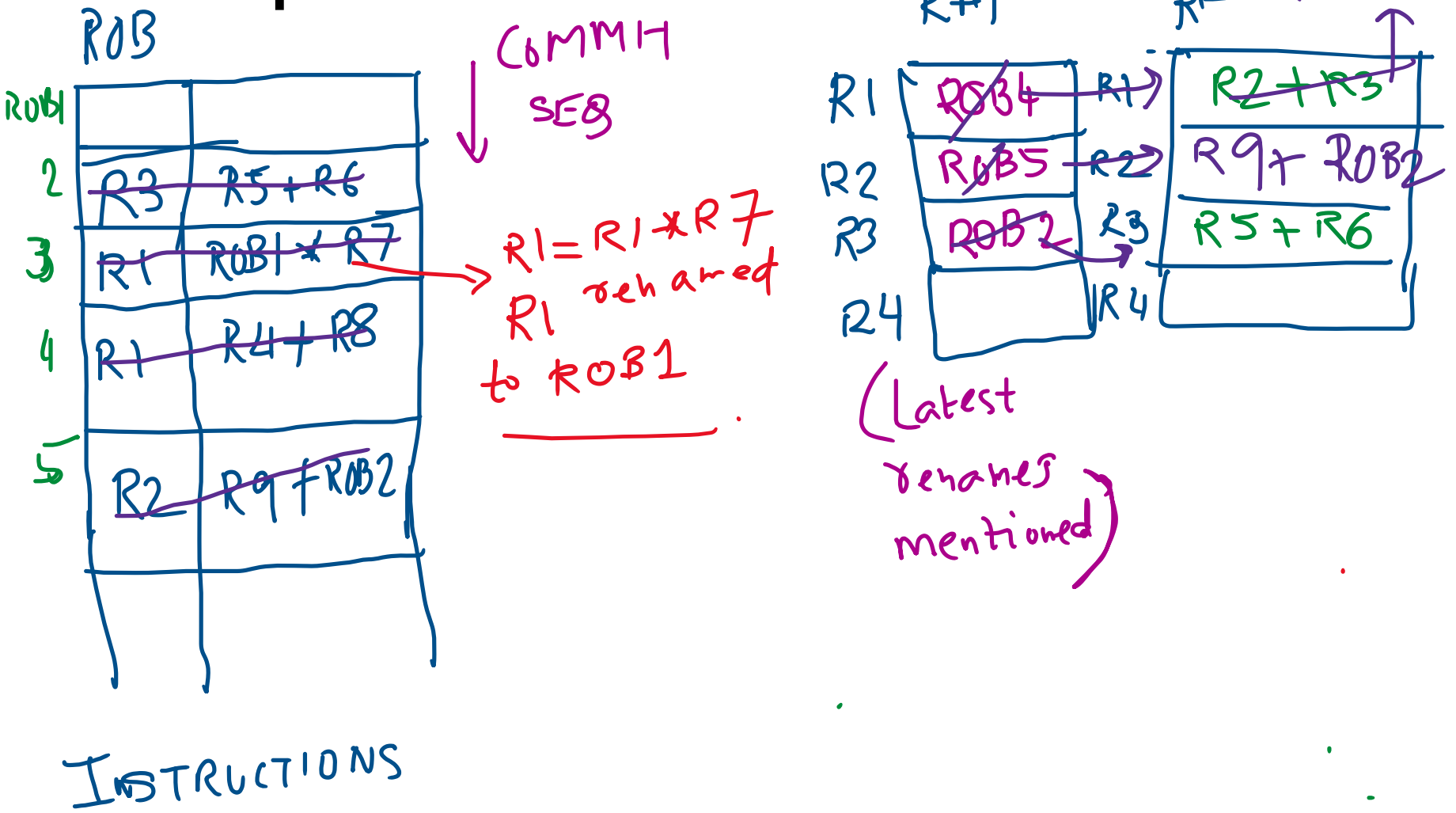
RAT Updates on Commit



RAT Updates on Commit



RAT Updates on Commit



Thank You!