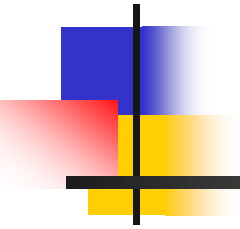
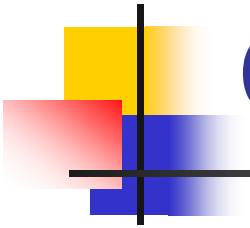


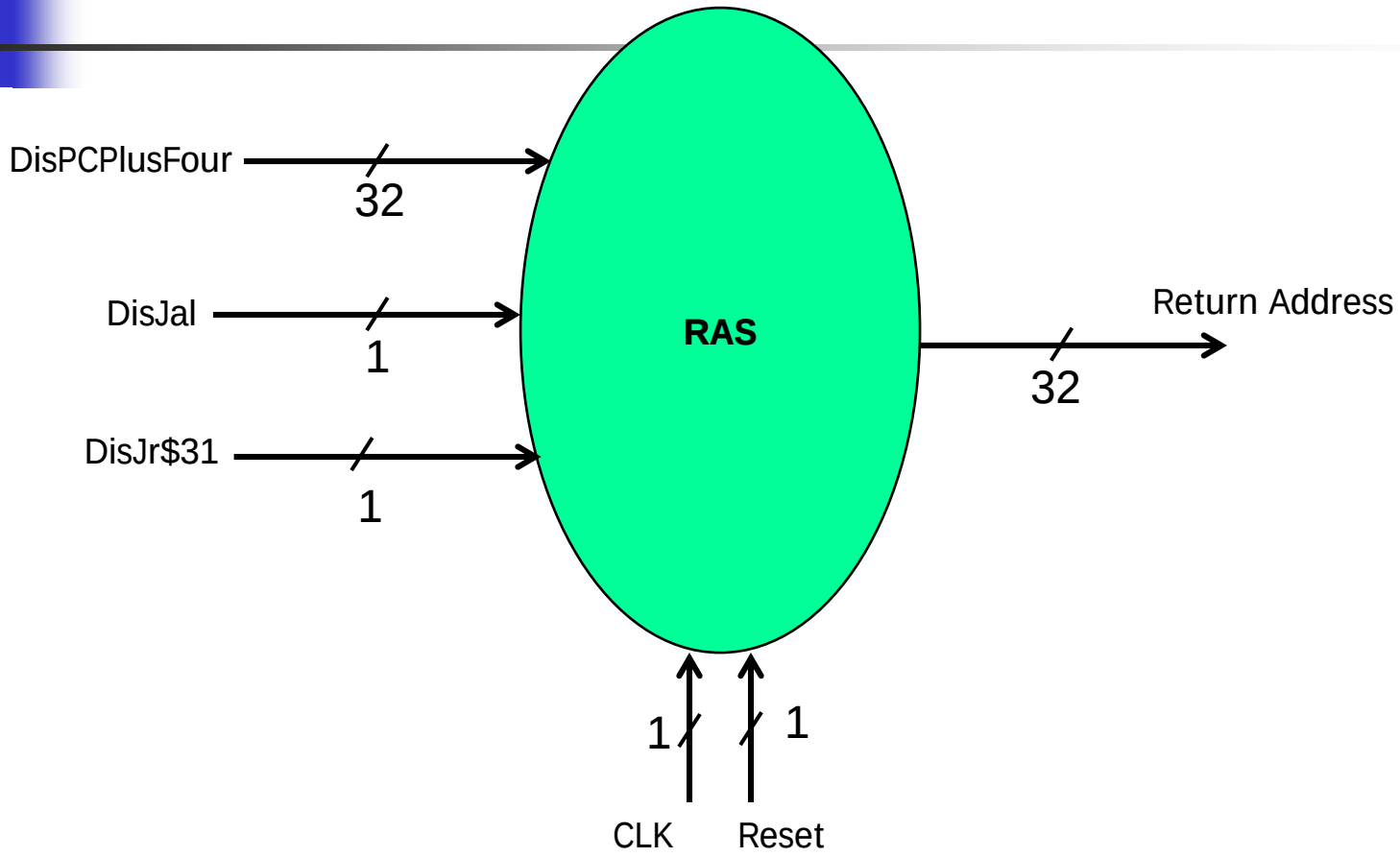
Return Address Stack (RAS)

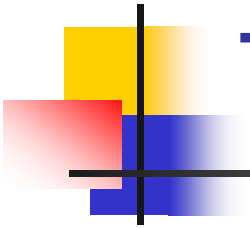


Overview

- RAS is 4 deep 32 bit wide stack.
- Located in Dispatch Stage.
- Stores ‘Return Address’ when JAL (Jump and Link) instruction is dispatched and Provides ‘Return Address’ when JR$31 (Return) instruction is dispatched.
- Contents may get corrupted/over-wrriten.
- Thus, ‘Return Address’ is just a prediction.

Pin-outs


DisPCPlusFour
32
DisJal
1
DisJr$31
1
RAS
Return Address
32
1
1
CLK
Reset

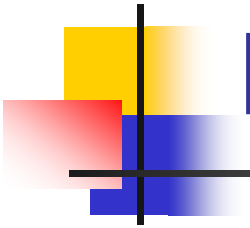


TOSP, TOSP+1

- Both 2 bit counters.

- TOSP: Top of the Stack Pointer.
- Points to last ‘Filled’ location in the Stack.
- Used to Pop latest data from stack.

- TOSP+1: Points to empty location immediately next to last filled location.
- Used to Push latest data onto stack.



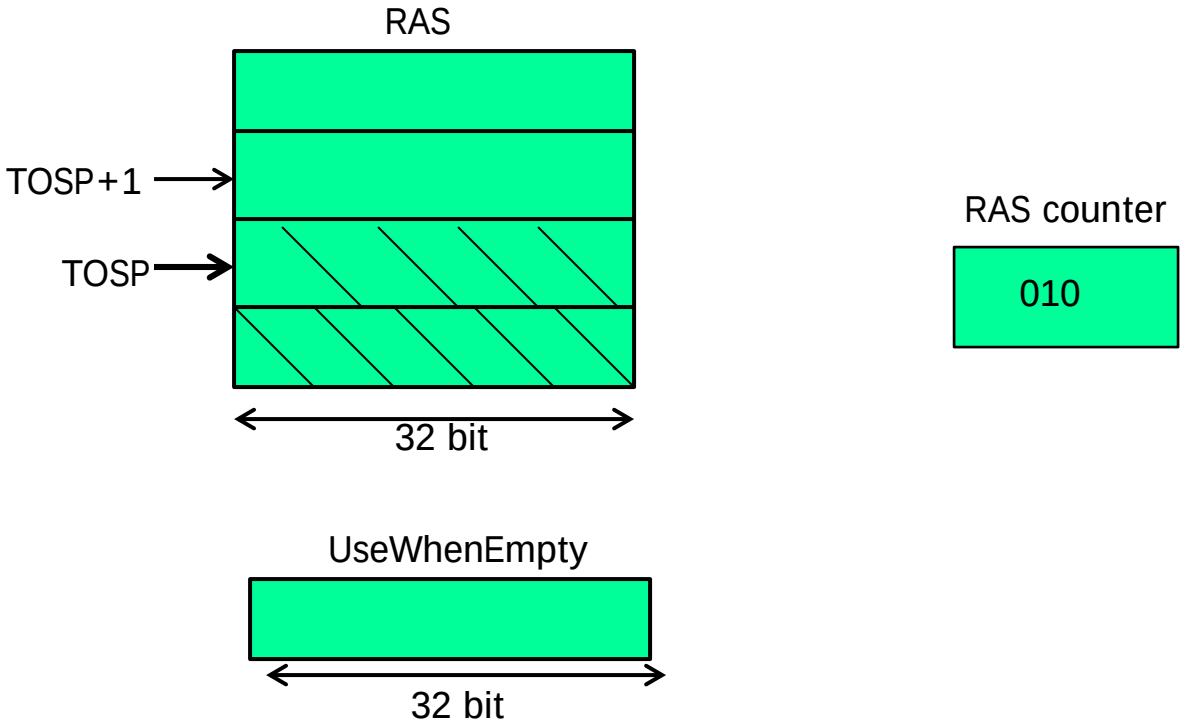
RAS Counter

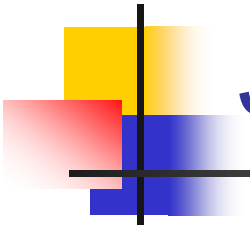
- 3 bit counter.
- Counts no. of filled location in the stack.
- Thus, “000” means RAS empty, “001” means 1 filled and so on. “100” means 4 filled and Full.
- Increments on PUSH(jal) and decrements when not empty on POP(jr$31). Counter saturates at “100”. Push at “100” won’t change the counter. Similarly it gets locked at “000”. So pop at “000” has no effect.

UseWhenEmpty Register

- 32 bit register.
- When we POP content out of RAS, we also store it in UseWhenEmpty register.
- Thus, when RAS becomes empty, this register contains last POPed data.
- When Empty, RAS provides this data as Return Addr. if it encounters JR before JAL. Thus RAS helps even when empty.
- Since RAS data is a prediction, it is not harmful if data is incorrect.

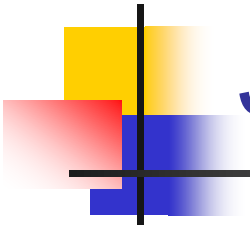
4-deep RAS organization





JAL

- Jump and Link
- Jumps to the address specified in OpCode (Function CALL) and stores PC+4 (Return Address) in $31.
- When Dispatch encounters JAL, it gives Jump address to IFQ as next fetch address, Pushes PC+4 onto RAS and issues the instruction to Integer Queue to store PC+4 in $31.



JR $31

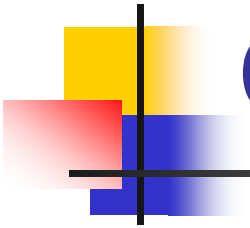
- Jump Register $31
- $31 is Poped from software stack.
- In the absence of RAS, JR requires to access contents of $31 and Jump to the address given. It issues inst. to Integer queue, requires it to complete execution and compute Jump address. Till then, we need to stall the pipeline.

JR $31 contd…

- With RAS, When dispatch encounters JR$31, it POPs address given by RAS and gives it to IFQ as next fetch address.
- It issues JR to Integer queue for computing actual Jump Address.
- The address given by RAS is a prediction and execution after JR$31 is speculative till actual contents of $31 are accessed.

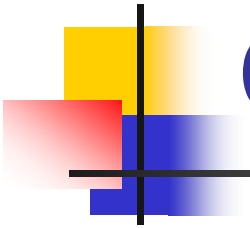
JR$31 contd...

- On Execution, contents of $31 are accessed and address provided by RAS and actual address are compared.
- If contents of $31 are different than Jump address, all the instructions Junior to JR$31 are flushed and actual contents of $31 are given as new fetch address.
- JR$31 retires from CDB itself.



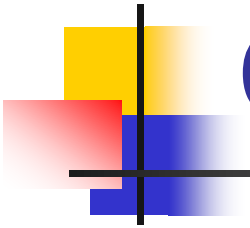
Overflow

- Since RAS is 4 deep, it overflows when a number of JALs dispatched, is 4 more than the number of JRs at any point of execution.
- Since RAS is circular buffer, we allow overwriting earlier data by latest data.
- For example, if RAS is full and Jal is dispatched, first location of RAS is overwritten, if we encounter another Jal before JR, second location will be overwritten and so on.



Overflow contd…

- Thus in case of overflow, we have valid data for last four PUSHes only.
- Since RAS counter saturates at 4, maximum 4 instructions can be POPed before RAS is empty.
- RAS still provides help for subsequent JR instructions when empty through UseWhenEmpty register, but it may or may not be correct.



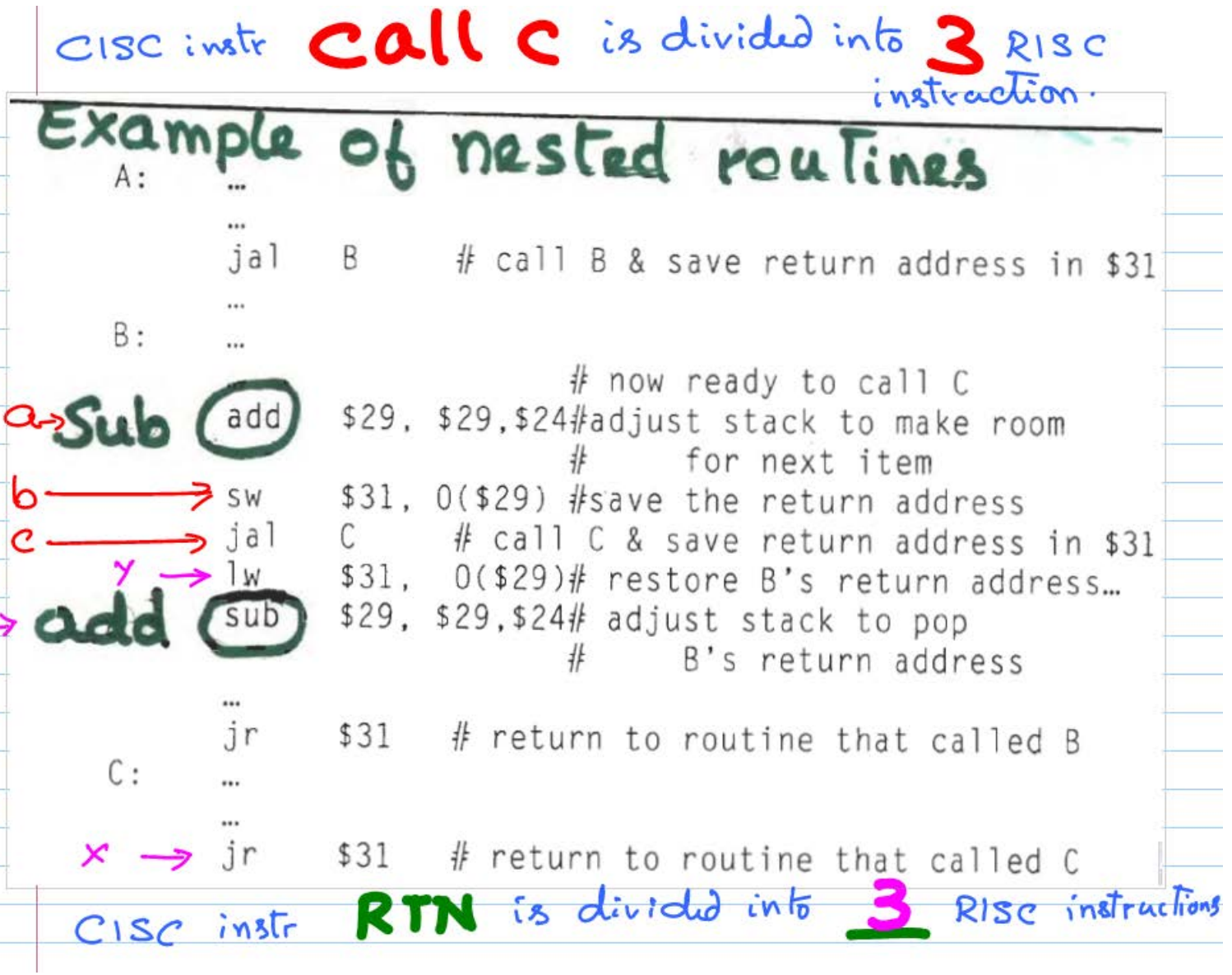
Corruption

- RAS can be corrupted by speculative execution of JAL and JR instructions due to predicted branch/predicted earlier JR$31. Such speculative JAL&JR may get flushed.
- We do not employ correction mechanism to RAS, but live with the error knowing that RAS will eventually repair itself.
- Also in case of overflow, we can only give correct return addresses to the last 4 calls and further help may be incorrect.

Software Stack

- We know $31 is'nt sufficient and a software stack (with fixed bottom) pointed to by $29 (compiler designated stack pointer) is maintained to store all the return addresses.
- In Assembly language programs, before Jal, we have two instructions to push the previous contents of $31 on to software stack and after Jal, we have two instructions to pop back the previous content of $31. Please see the next slide from Ch#3 of EE457. Return address predicted by RAS is verified (as correct or incorrect) when Jr $31 goes on CDB.

Software Stack continued

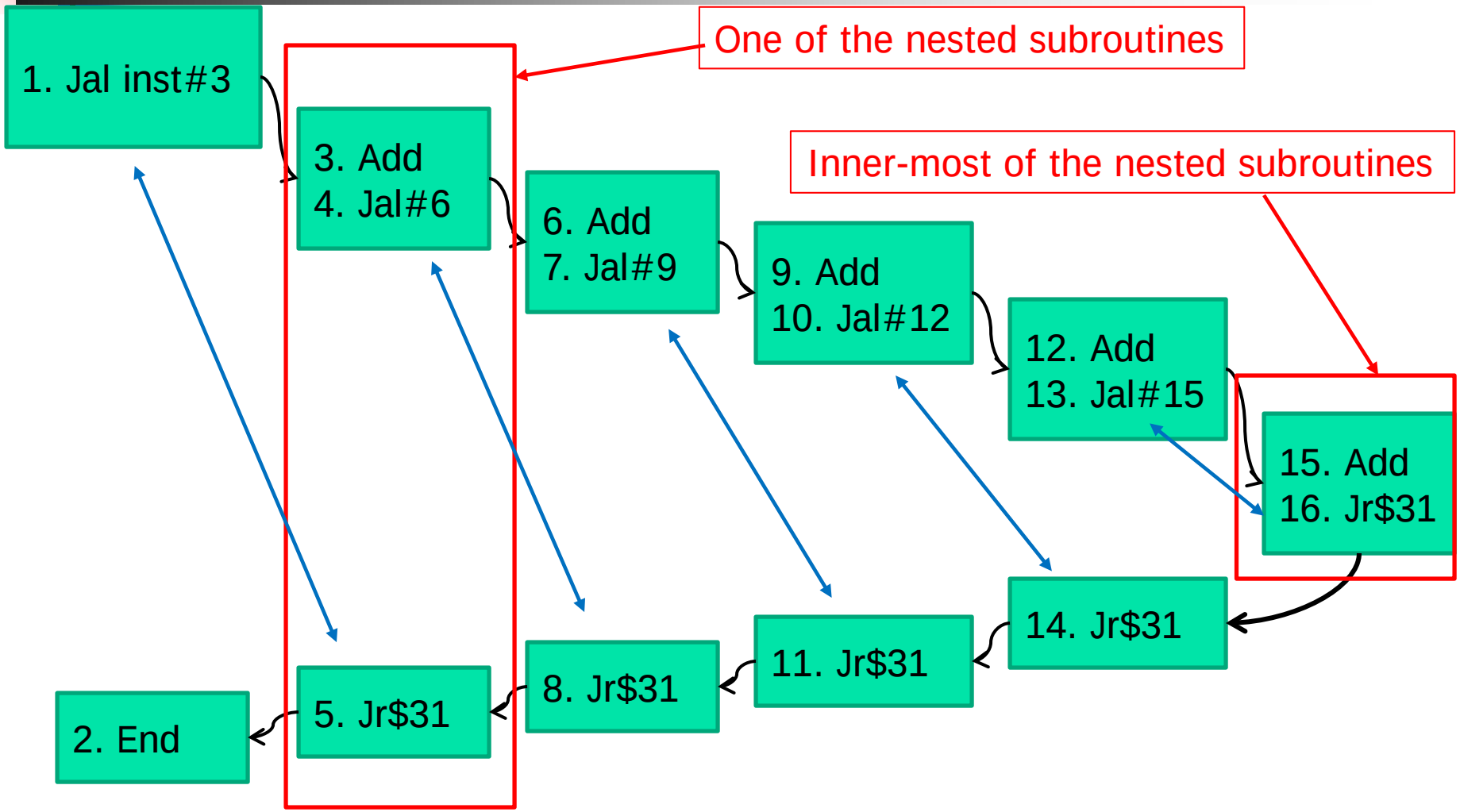


Example (first look at the next page to understand the following)

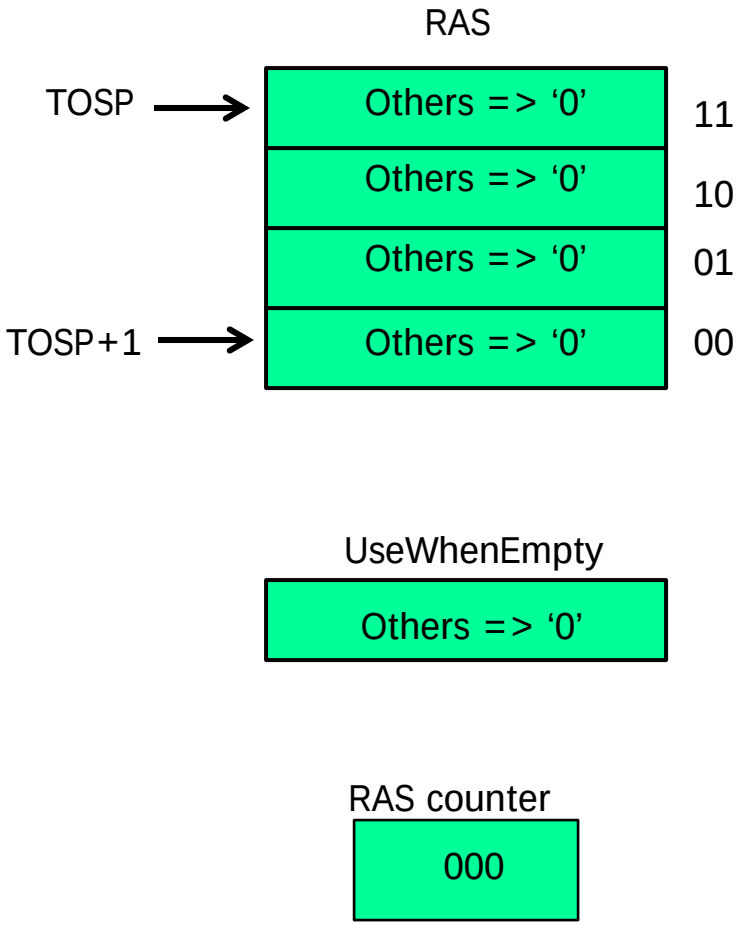
1. Jal inst#3
2. End
3. Add $2, $2, $2
4. Jal inst#6
5. JR$31
6. Add $3, $3, $3
7. Jal inst#9
8. JR$31
9. Add $4, $4, $4
10. Jal inst#12
11. JR$31
12. Add $5, $5, $5
13. Jal inst#15
14. JR$31
15. Add $6, $6, $6
16. JR$31

Pictorial View

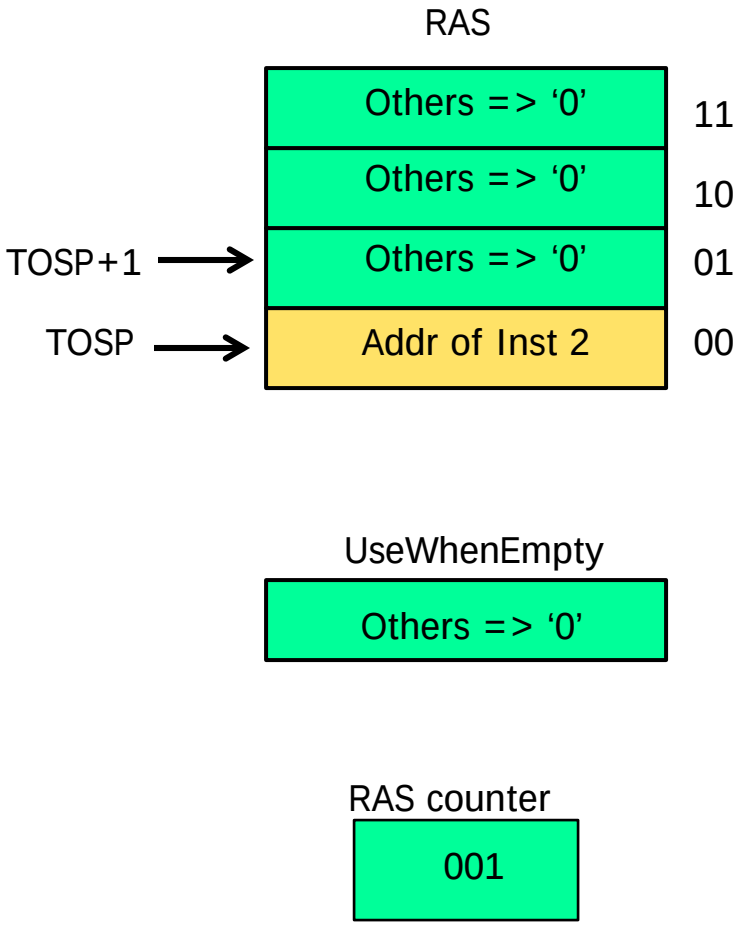
5 Jal's and 5 Jr's; RAS is only 4 deep



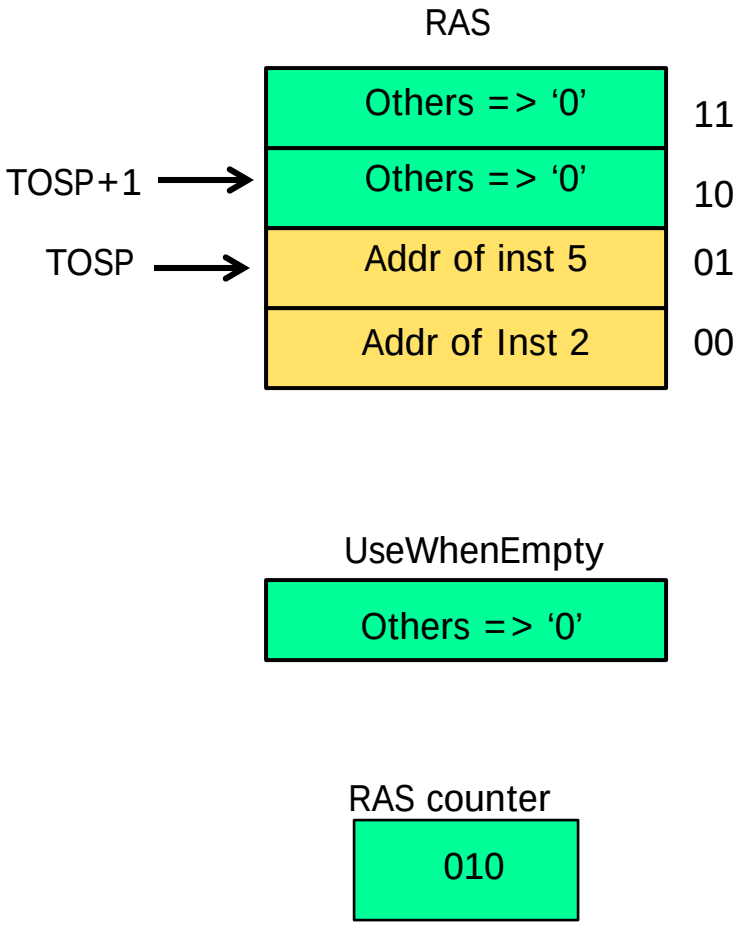
Execution



On Reset


RAS
Others => '0'
Others => '0'
Others => '0'
Addr of Inst 2
11
10
01
00
TOSP+1
TOSP
UseWhenEmpty
Others => '0'
RAS counter
001

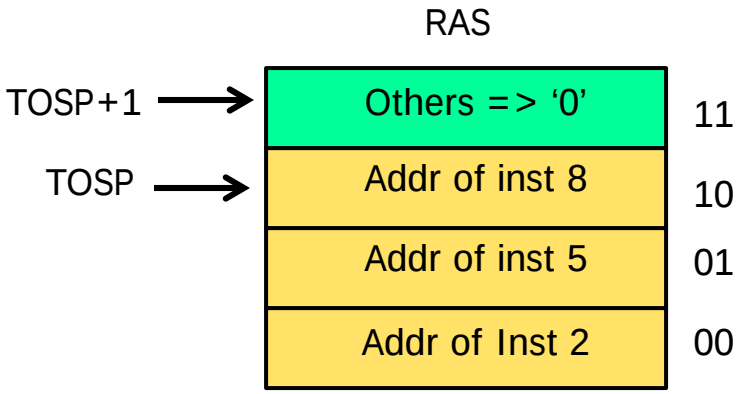
1) Jal inst#3

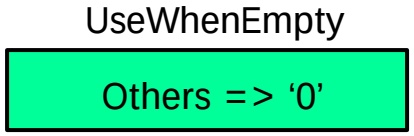


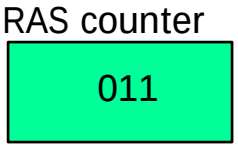
1) Jal inst#3

3) Add $2, $2, $2

4) Jal inst#6







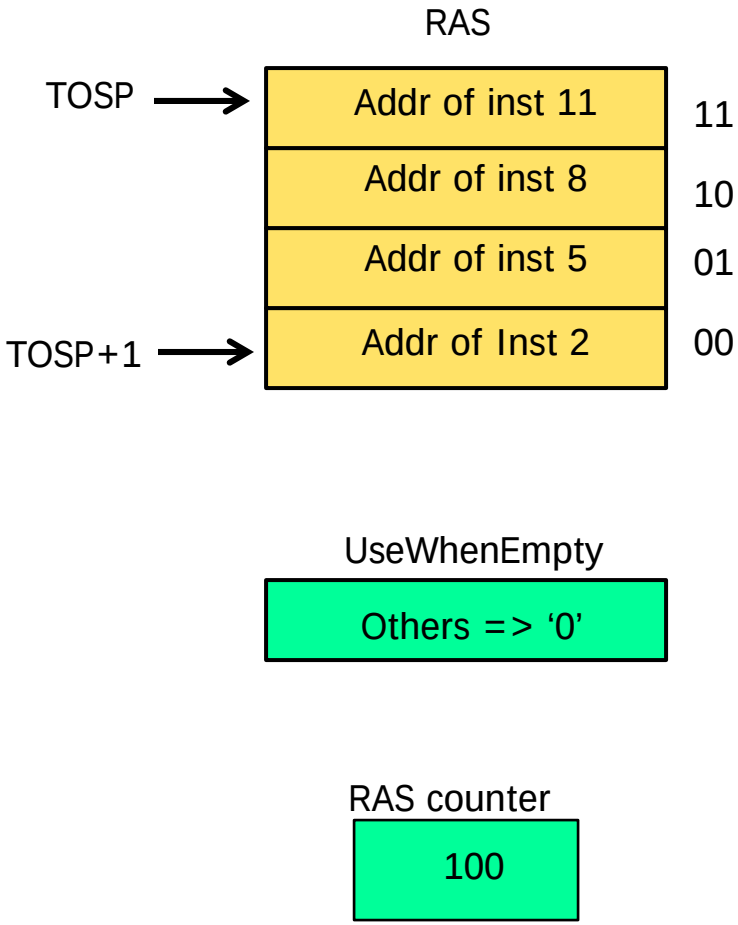
1) Jal inst#3

3) Add $2, $2, $2

4) Jal inst#6

6) Add $3, $3, $3

7) Jal inst#9



1) Jal inst#3

3) Add $2, $2, $2

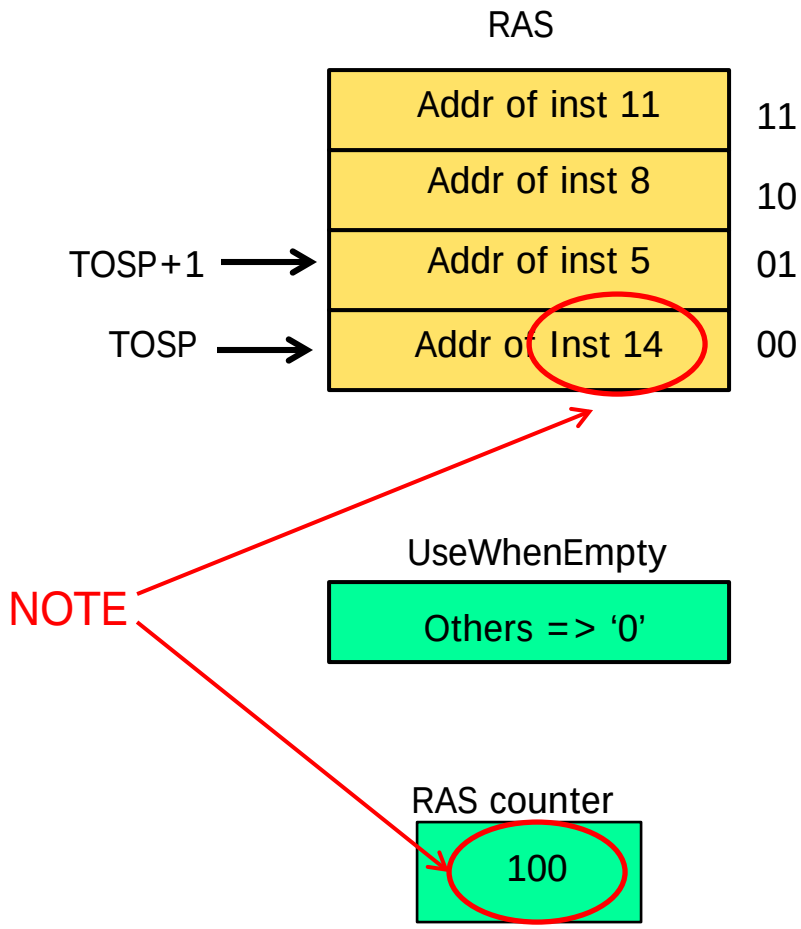
4) Jal inst#6

6) Add $3, $3, $3

7) Jal inst#9

9) Add $4, $4, $4

10) Jal inst#12



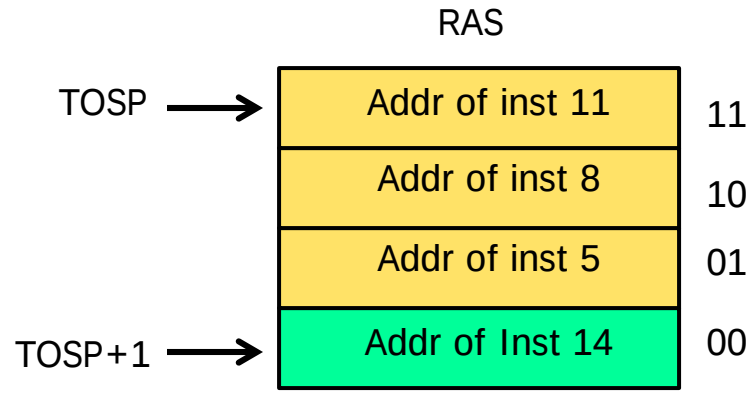
1) Jal inst#3

3) Add $2, $2, $2
4) Jal inst#6

6) Add $3, $3, $3
7) Jal inst#9

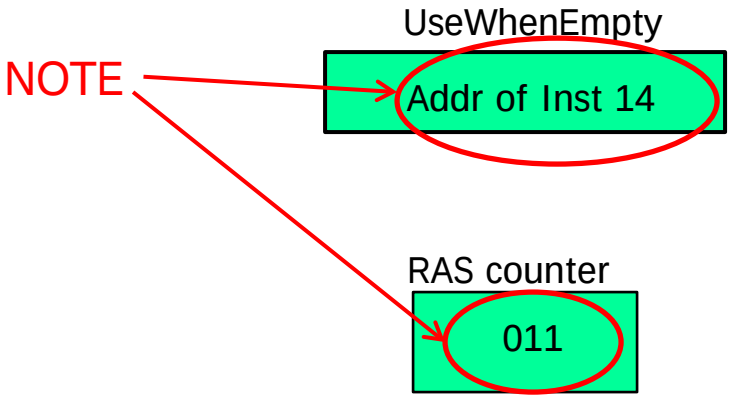
9) Add $4, $4, $4
10) Jal inst#12

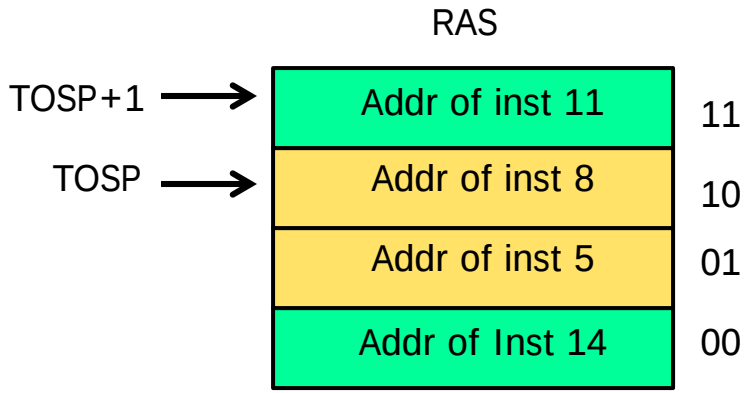
12) Add $5, $5, $5
13) Jal inst#15



15) Add $6, $6, $6

16) JR$31

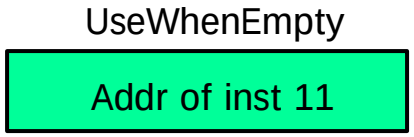


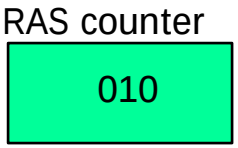


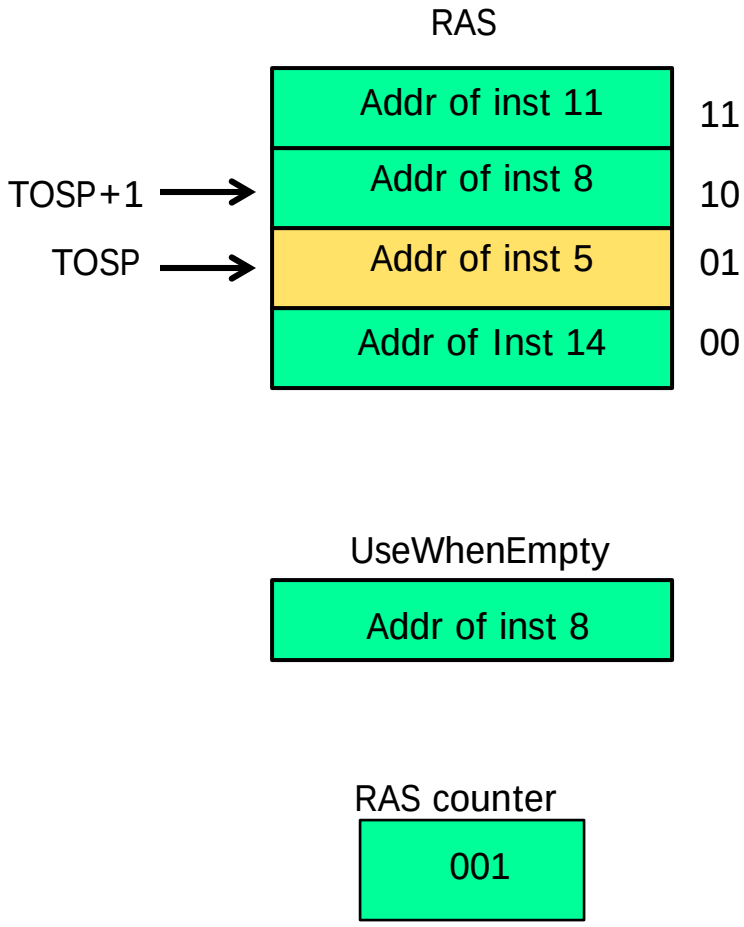
15) Add $6, $6, $6

16) JR$31

14) JR$31





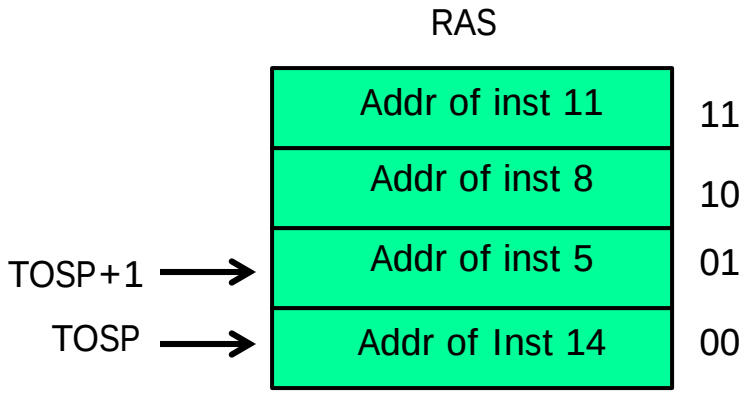


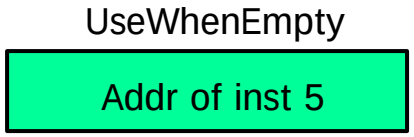
15) Add $6, $6, $6

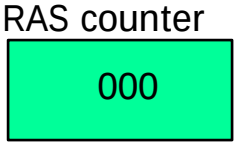
16) JR$31

14) JR$31

11) JR$31







15) Add $6, $6, $6

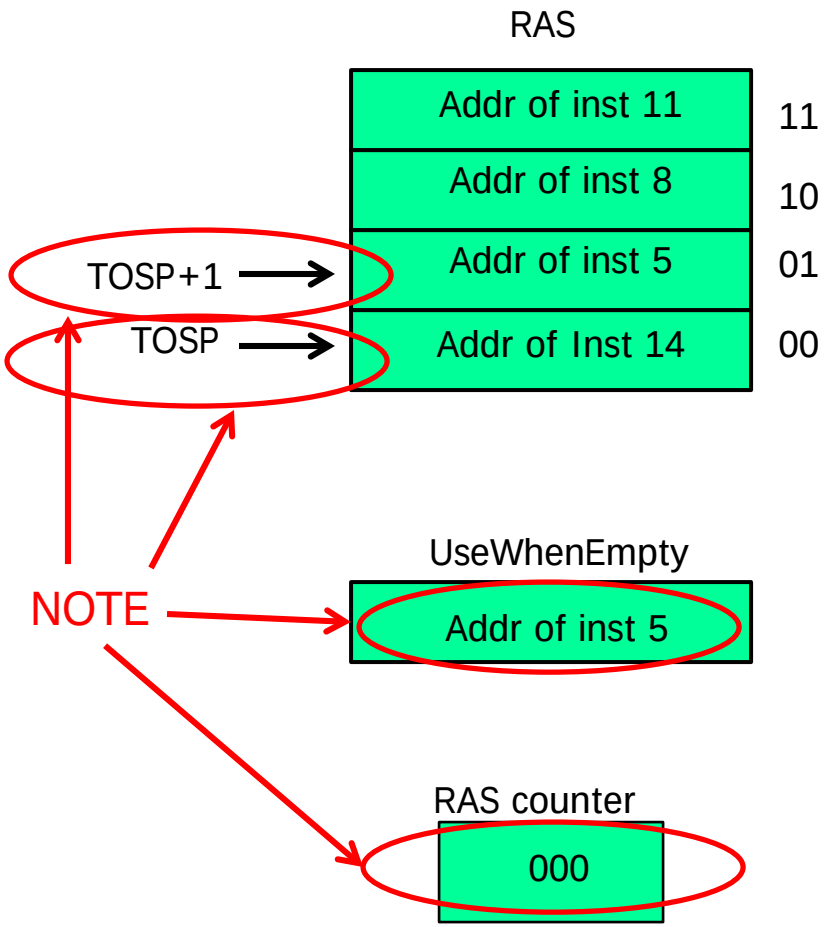
16) JR$31

14) JR$31

11) JR$31

8) JR$31

At this point, since RAS is empty, note that we provide incorrect help from UseWhenEmpty register and thus we will NOT fetch Inst 2 as desired in the next clock, but again inst 5 and so on... and after the first inst 5 is actually executed, we will flush all additional unwanted inst 5's and we will fetch inst 2.



15) Add $6, $6, $6

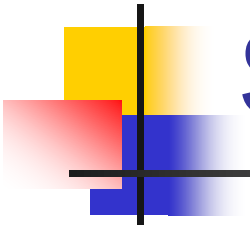
16) JR$31

14) JR$31

11) JR$31

8) JR$31

5) JR$31



Summary

- RAS is a 4-deep circular stack which stores return addresses for JAL (Function calls) and provides return addresses when JR$31 (Return from subroutine) needs it.
- RAS speeds up execution by providing return addr. from dispatch stage instead of from execution stage and hence avoids stalling dispatch.
- It is a prediction and may be wrong.
- When JR$31 completes execution and comes on CDB and announces that it is predicted wrong, all junior instructions (based on ROB tags) will get flushed. The JR $31 also announces (from CDB) the correct return address. IFQ starts fetching from the correct return address now.