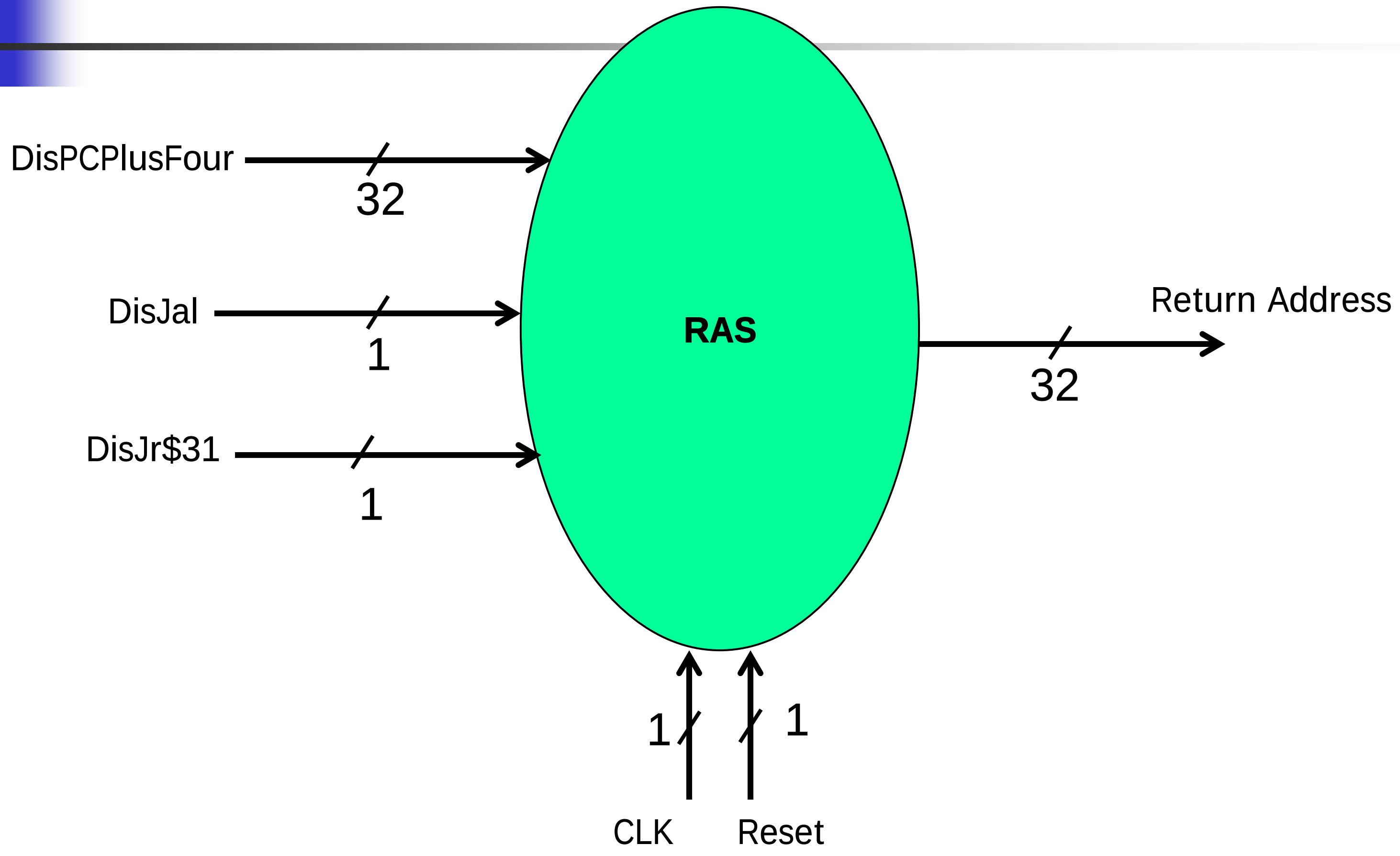
Return Address Stack (RAS)

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.

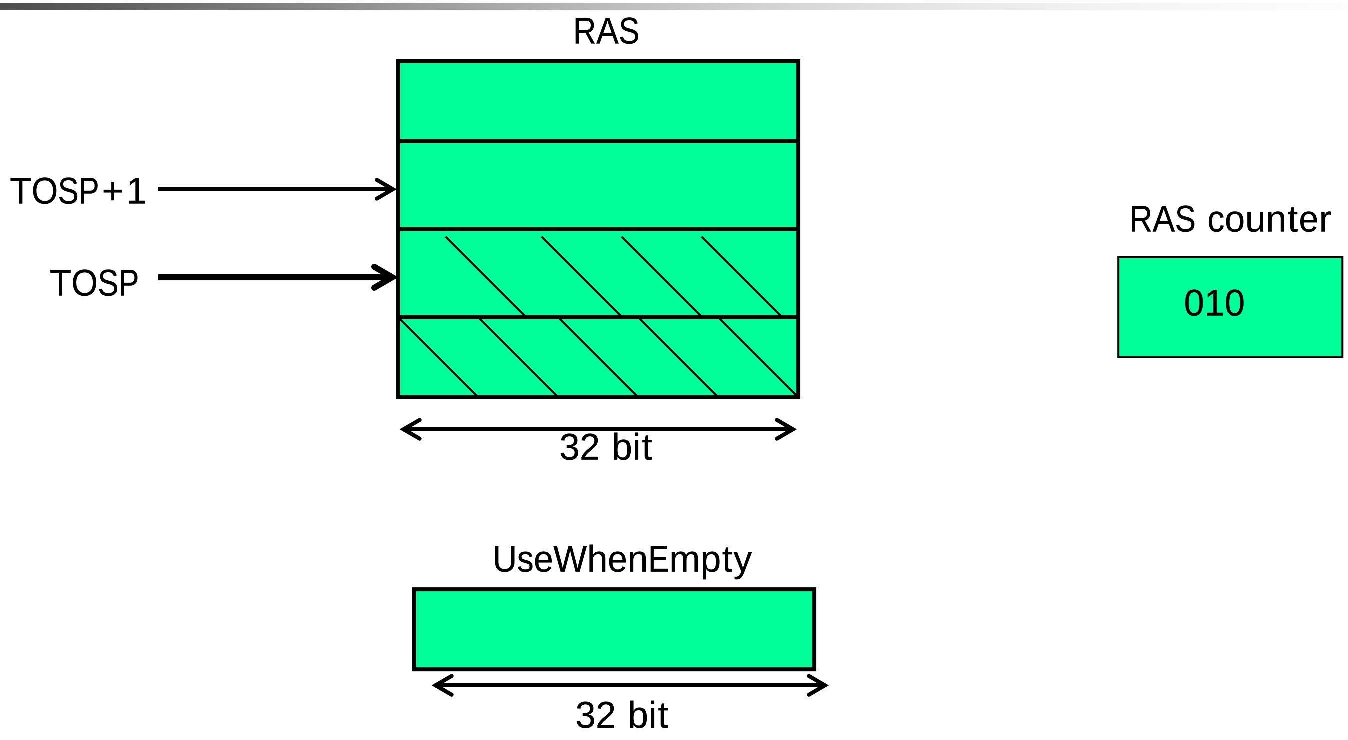
RAS Counter

- 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.

Complete Picture



RAS

TOSP+1

TOSP

32 bit

RAS counter

010

UseWhenEmpty

32 bit

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/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 it is a speculation.
- Also in case of overflow, we can only give correct help to latest 4 calls and further help may be incorrect.

Software Stack

- Usually $31 is not a register but a stack, with fixed bottom and stores all the return addresses.
- To emulate this, We use $29, for special purpose. It serves as pointer to software stack. Before dispatching JAL, we use following 2 instructions:
- addi $29, $29, -4
- sw $31,0($29)

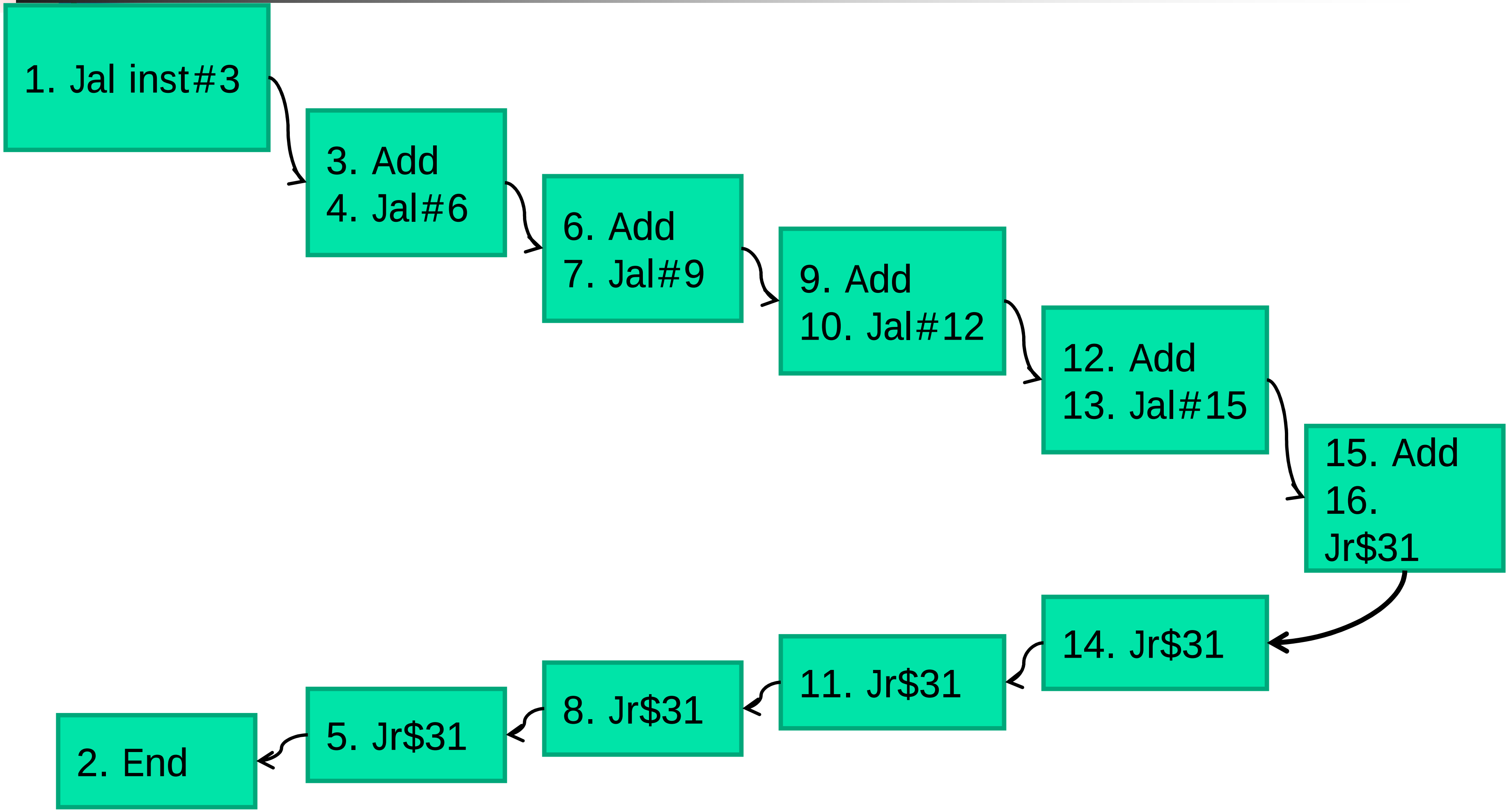
Software Stack cntd....

- Before dispatching JR$31, we use following two instructions:
- lw $31,0($29)
- addi $29, $29, 4
- So in the instruction trace we have to pad up JAL and JR$31 with above instructions to ensure that we store right return address in case RAS gives incorrect help.

Example

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



1. Jal inst#3

3. Add
4. Jal#6

6. Add
7. Jal#9

9. Add
10. Jal#12

12. Add
13. Jal#15

15. Add
16. Jr$31

14. Jr$31

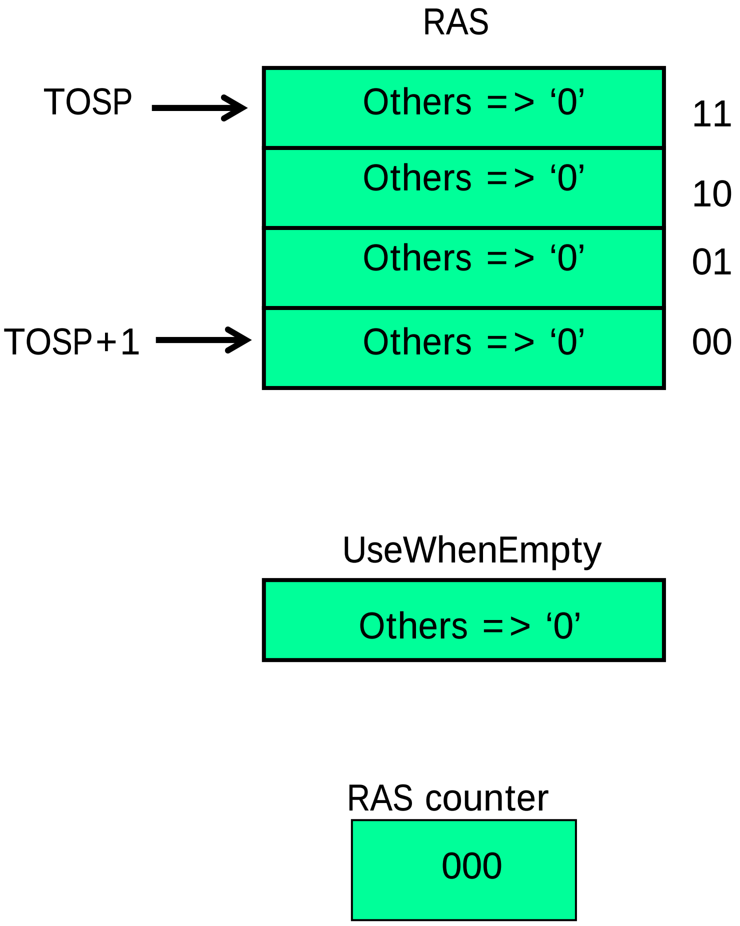
11. Jr$31

8. Jr$31

5. Jr$31

2. End

Execution



RAS

TOSP → Others => ‘0’ 11

Others => ‘0’ 10

Others => ‘0’ 01

TOSP+1 → Others => ‘0’ 00

UseWhenEmpty

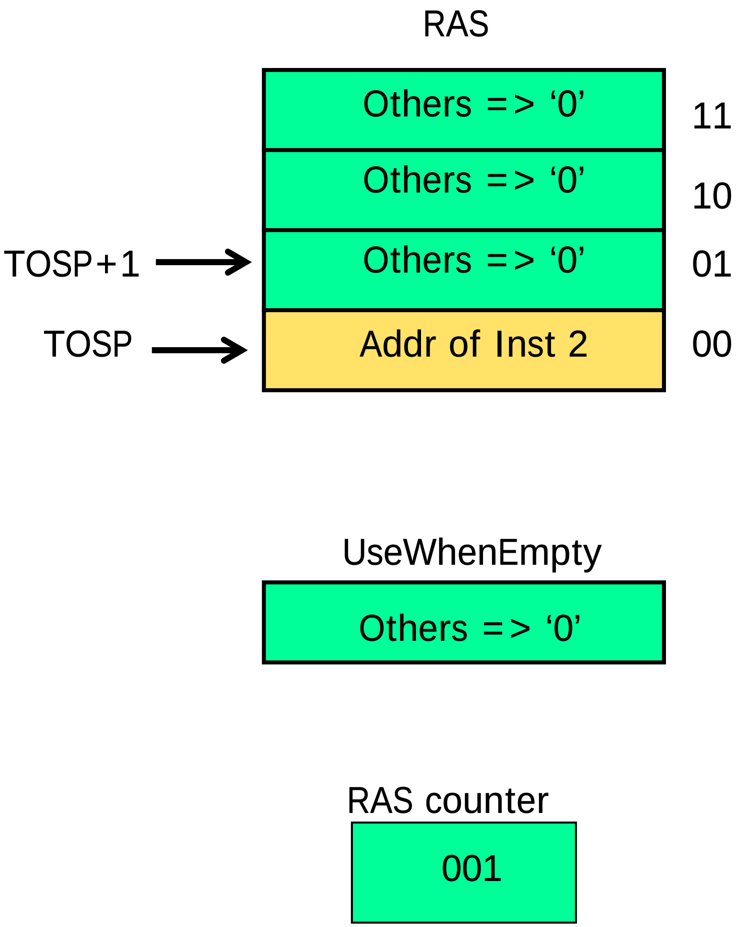
Others => ‘0’

RAS counter

000

On Reset

1) Jal inst#3



RAS

Others => ‘0’ 11

Others => ‘0’ 10

TOSP+1 → Others => ‘0’ 01

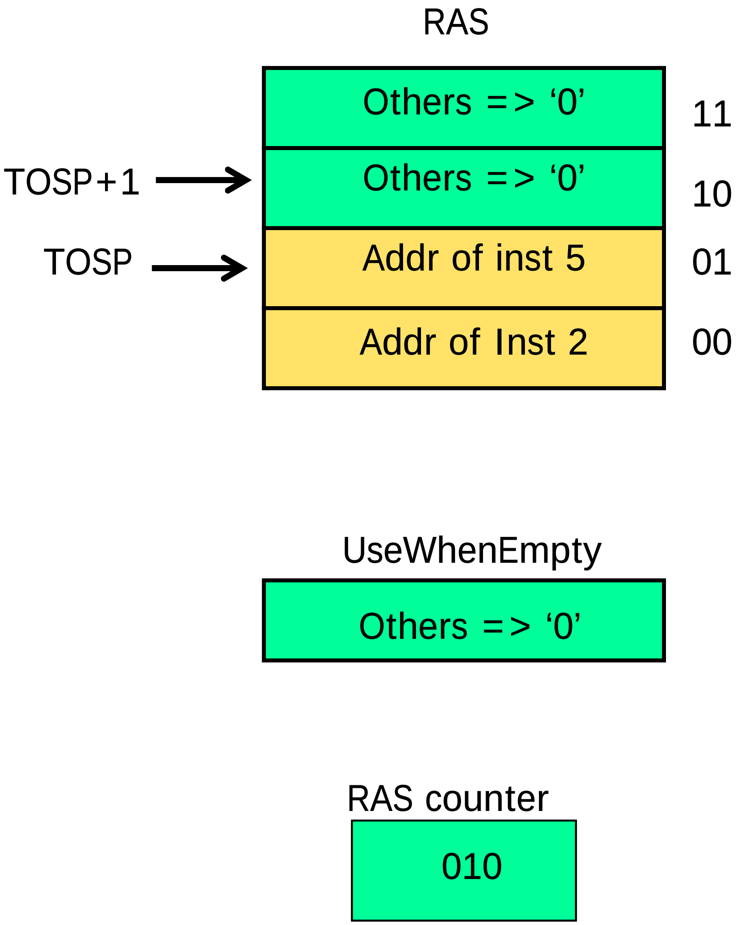
TOSP → Addr of Inst 2 00

UseWhenEmpty

Others => ‘0’

RAS counter

001



RAS

	Others => ‘0’	11
TOSP+1 →	Others => ‘0’	10
TOSP →	Addr of inst 5	01
	Addr of Inst 2	00

UseWhenEmpty

Others => ‘0’

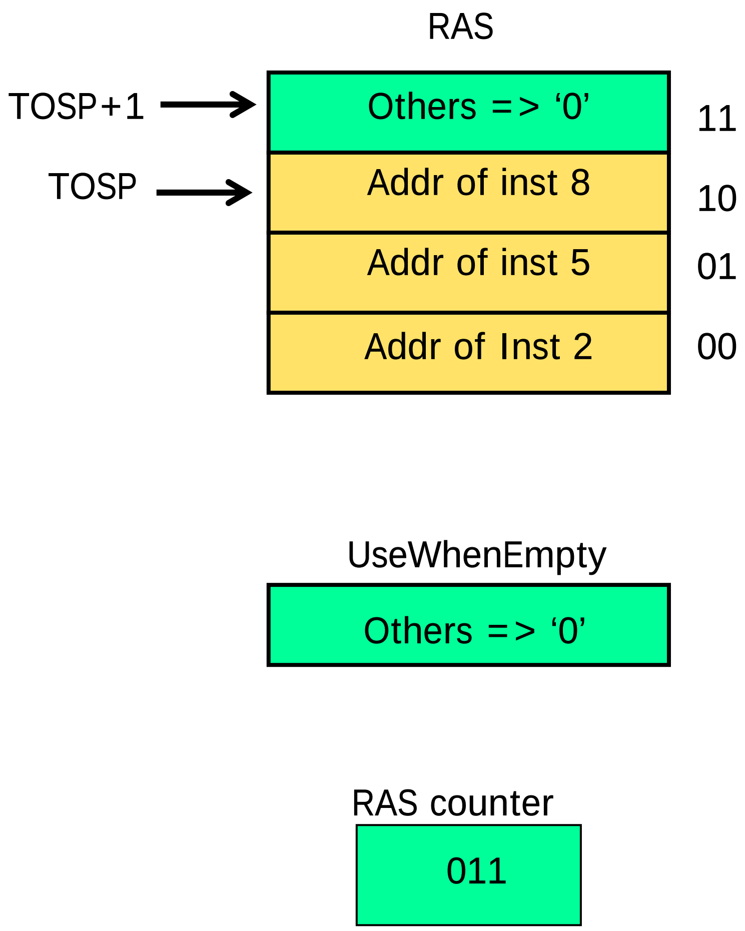
RAS counter

010

1) Jal inst#3

3) Add $2, $2, $2

4) Jal inst#6



RAS

TOSP+1

TOSP

Others => ‘0’

Addr of inst 8

Addr of inst 5

Addr of Inst 2

11

10

01

00

UseWhenEmpty

Others => ‘0’

RAS counter

011

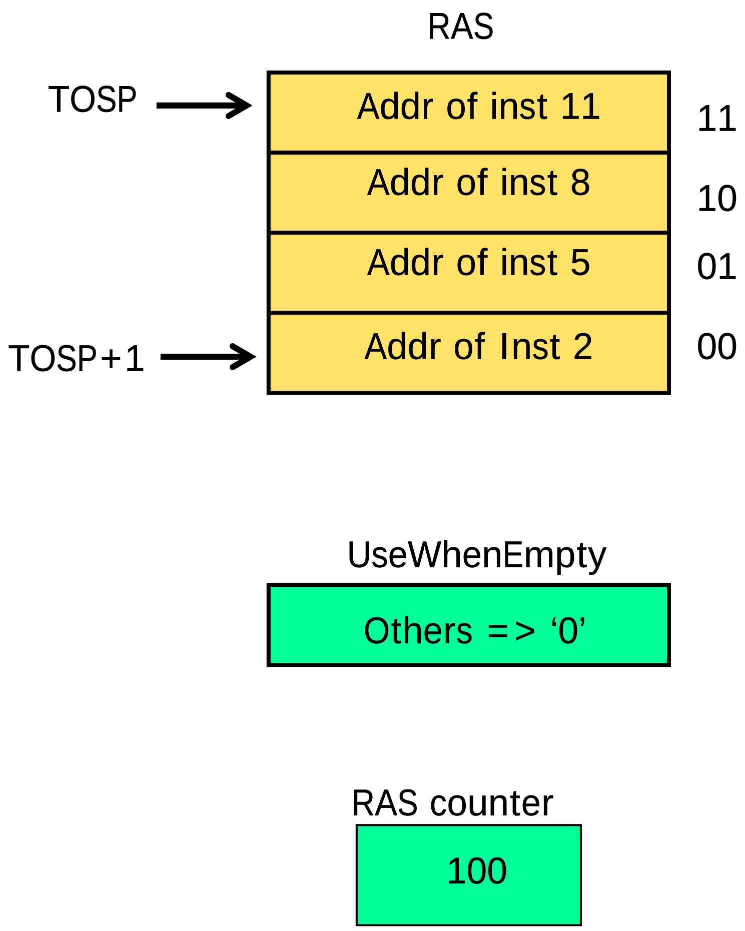
1) Jal inst#3

3) Add $2, $2, $2

4) Jal inst#6

6) Add $3, $3, $3

7) Jal inst#9



RAS

TOSP

Addr of inst 11

11

Addr of inst 8

10

Addr of inst 5

01

TOSP+1

Addr of Inst 2

00

UseWhenEmpty

Others => ‘0’

RAS counter

100

1) Jal inst#3

3) Add $2, $2, $2

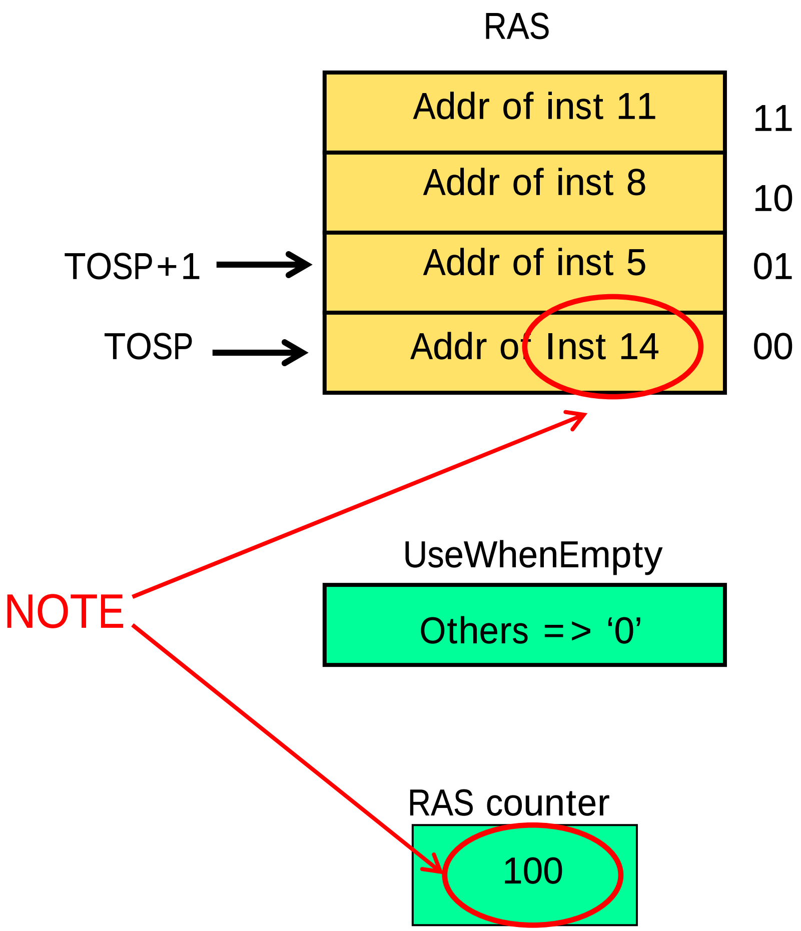
4) Jal inst#6

6) Add $3, $3, $3

7) Jal inst#9

9) Add $4, $4, $4

10) Jal inst#12



RAS

RAS	
Addr of inst 11	11
Addr of inst 8	10
Addr of inst 5	01
Addr of Inst 14	00

TOSP+1

TOSP

NOTE

UseWhenEmpty

Others => ‘0’

RAS counter

100

1) Jal inst#3

3) Add $2, $2, $2

4) Jal inst#6

6) Add $3, $3, $3

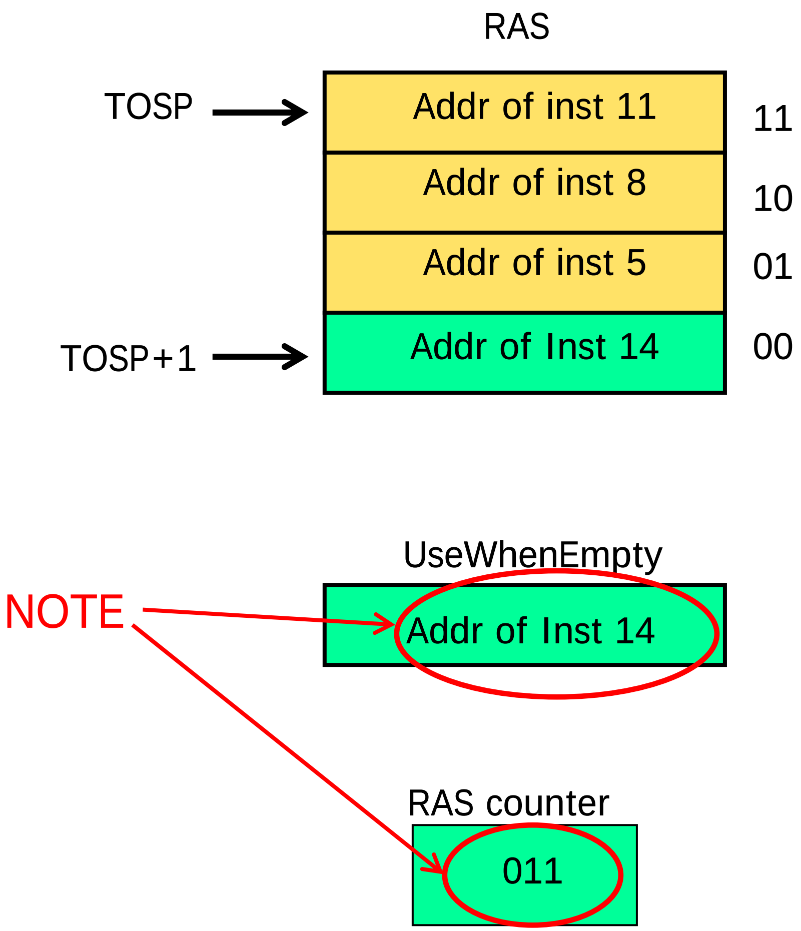
7) Jal inst#9

9) Add $4, $4, $4

10) Jal inst#12

12) Add $5, $5, $5

13) Jal inst#15



RAS

TOSP

Addr of inst 11	11
Addr of inst 8	10
Addr of inst 5	01
Addr of Inst 14	00

TOSP+1

UseWhenEmpty

Addr of Inst 14

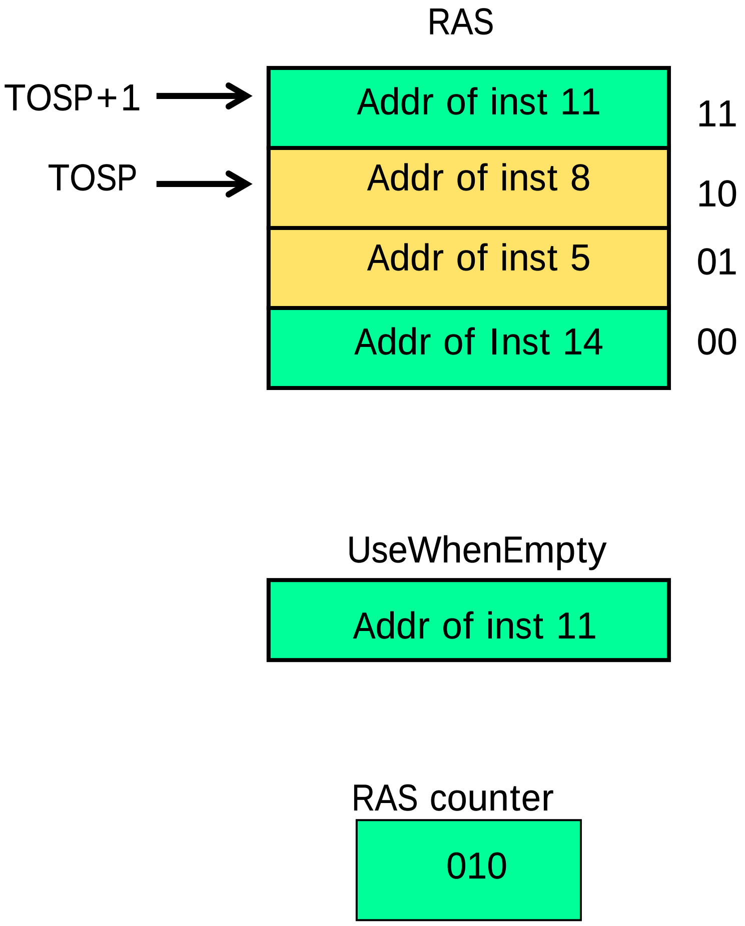
NOTE

RAS counter

011

15) Add $6, $6, $6

16) JR$31



RAS

TOSP+1 → Addr of inst 11 — 11

TOSP → Addr of inst 8 — 10

Addr of inst 5 — 01

Addr of Inst 14 — 00

UseWhenEmpty

Addr of inst 11

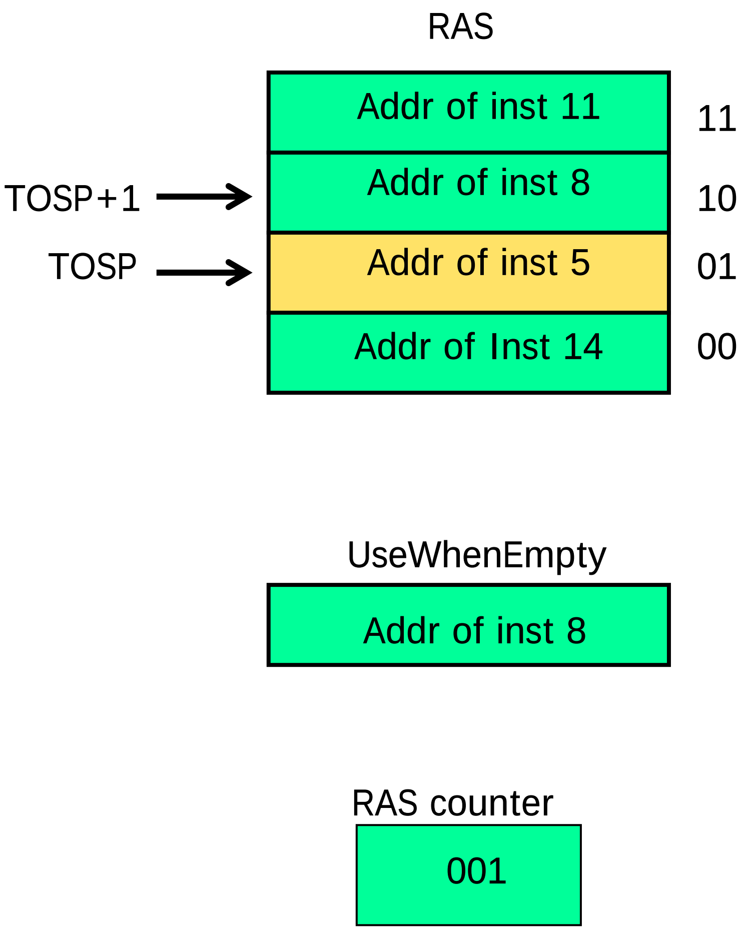
RAS counter

010

15) Add $6, $6, $6

16) JR$31

14) JR$31



RAS

TOSP+1

TOSP

Addr of inst 11	11
Addr of inst 8	10
Addr of inst 5	01
Addr of Inst 14	00

UseWhenEmpty

Addr of inst 8

RAS counter

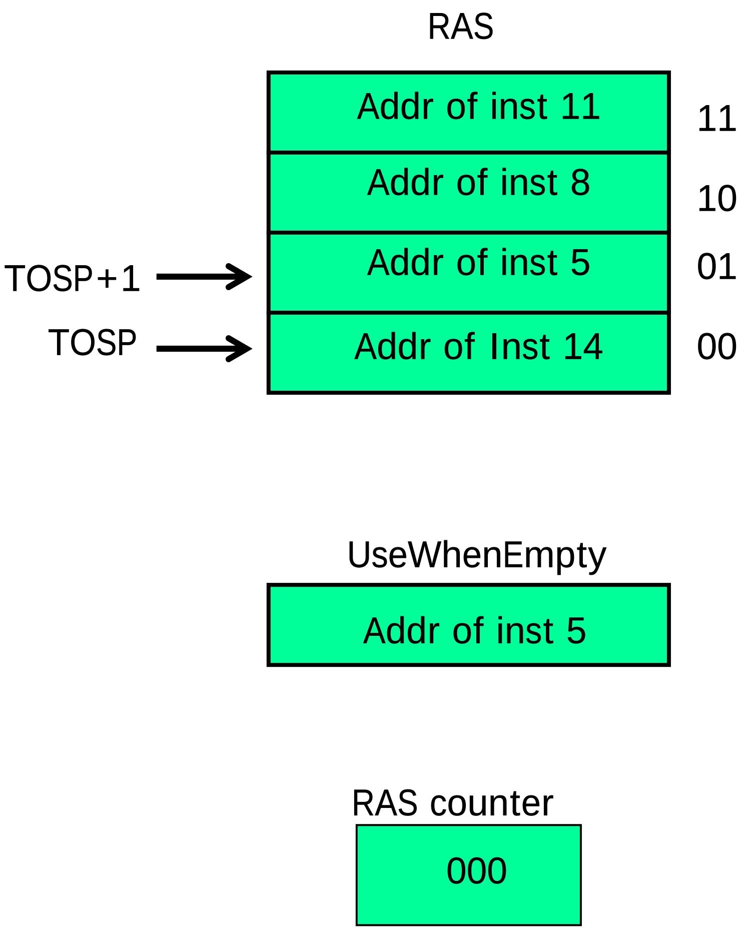
001

15) Add $6, $6, $6

16) JR$31

14) JR$31

11) JR$31



RAS

Addr of inst 11	11
Addr of inst 8	10
Addr of inst 5	01
Addr of Inst 14	00

TOSP+1

TOSP

UseWhenEmpty

Addr of inst 5

RAS counter

000

15) Add $6, $6, $6

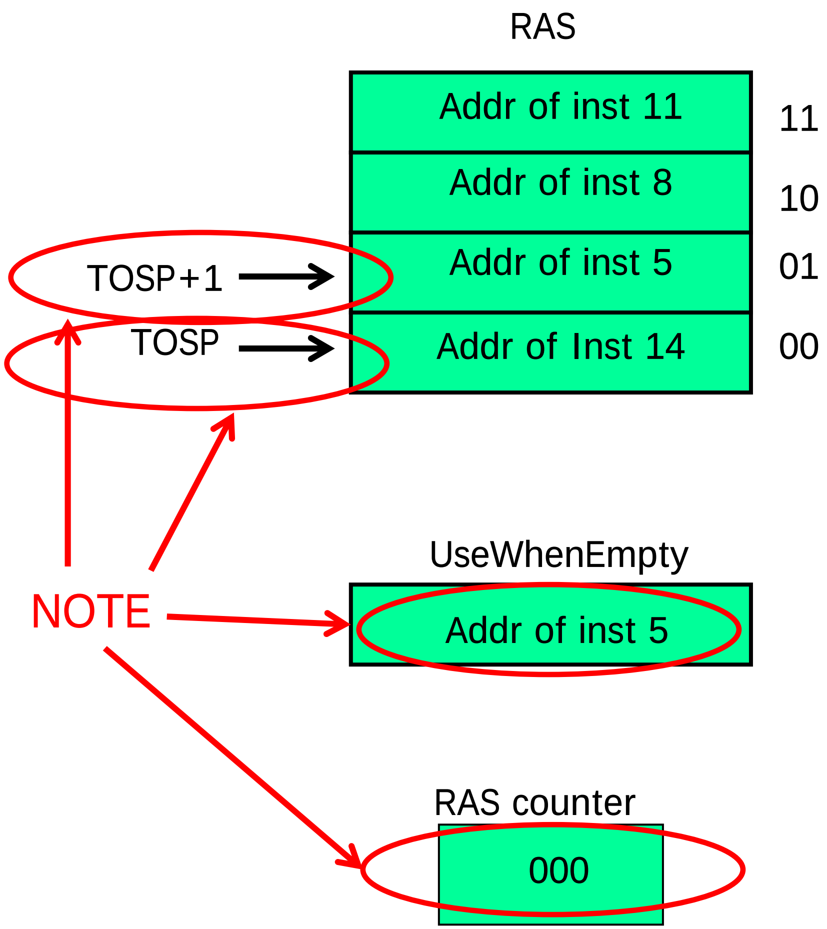
16) JR$31

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 instruction is actually executed, we will fetch from inst 2.



RAS

Addr of inst 11 — 11

Addr of inst 8 — 10

TOSP+1 → Addr of inst 5 — 01

TOSP → Addr of Inst 14 — 00

NOTE

UseWhenEmpty

Addr of inst 5

RAS counter

000

15) Add $6, $6, $6

16) JR$31

14) JR$31

11) JR$31

8) JR$31

5) JR$31

Summary

- RAS is 4 deep circular stack which stores return address for JAL (Function calls) and provides when JR$31 (Return) needs it.
- RAS speeds up execution by providing return addr. from dispatch stage instead of execution stage and hence avoids stalling.
- It is a prediction and may be wrong which initiates flush. In that case, actually jump addr. is provided by execution of JR$31.