Copy Free Check-pointing (CFC) without walking in ROB (Phase 1)

CFC Reference

Towards a Viable Out-of-Order Soft Core: Copy-free, Checkpointed Register Renaming

Design Ideas and figures are borrowed from the above paper.

Pre-req

- Knowledge of the following is assumed:
  - Tomasulo processor with Physical Register File (PRF), Retirement Register Alias Table (RRAT) and Frontend RAT (FRAT)
  - ROB Walking on branch misprediction (for Phase 2 only)

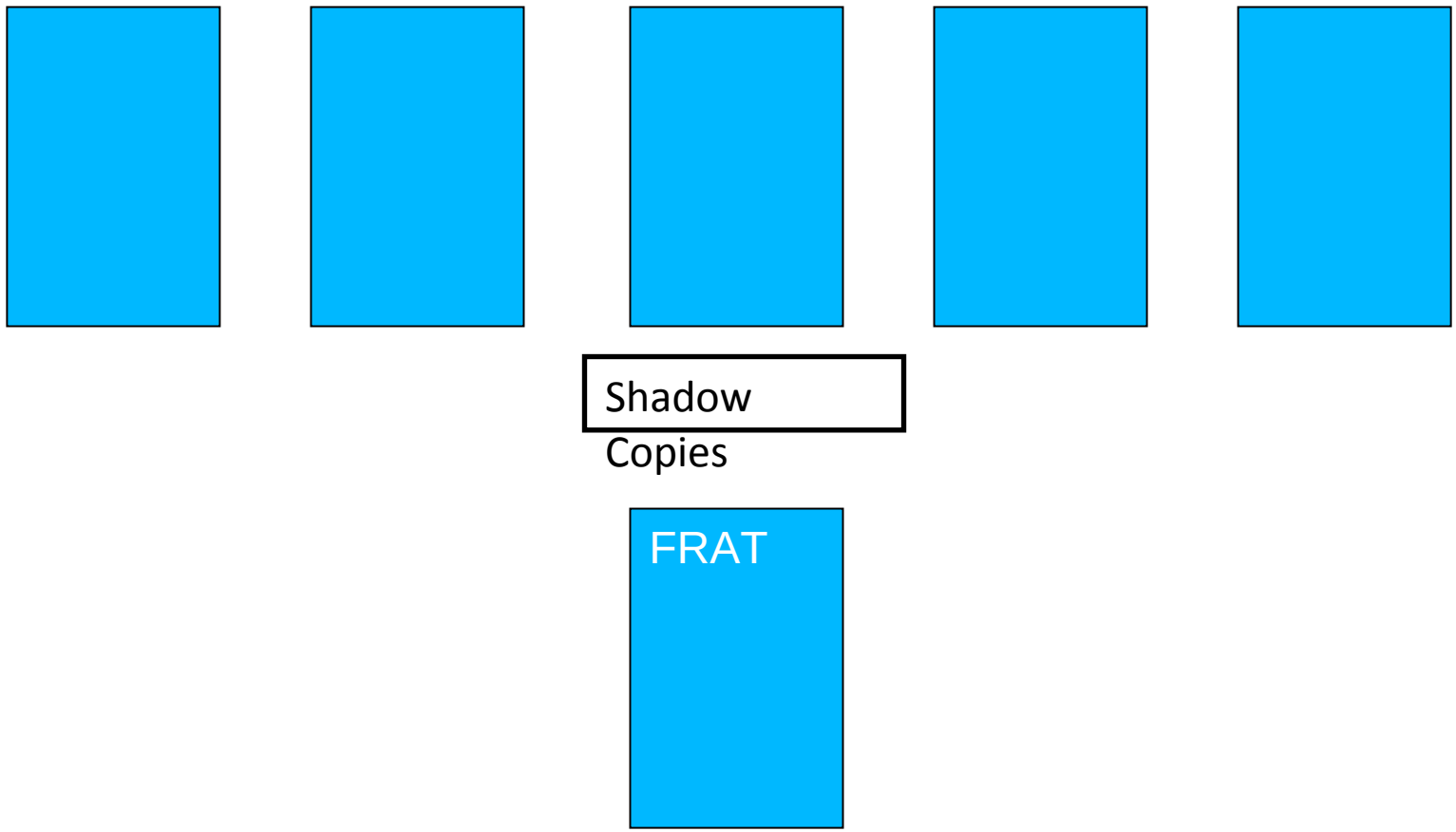
Motivation

- Do not want ROB walking on frequently mispredicted branches since it takes multiple clocks
- Checkpointing the state on disptach of these branches and restoring it to that state when a branch is mispredicted is a single-clock solution.
- We do not do ROB walking since it is complicated, and only do checkpointing

Motivation

- For any processor, state of an execution is the architectural register values and PC
- Since in our design, we have a PRF and an FRAT, it suffices to restore the FRAT and FRL(Free Register List) state to when the branch was dispatched.
- Can be done by keeping shadow copies of FRAT and FRL and copying the entire shadow copy onto the FRAT and FRL when branch is mispredicted
- However, large array copy in FPGAs would consume excessive resources.

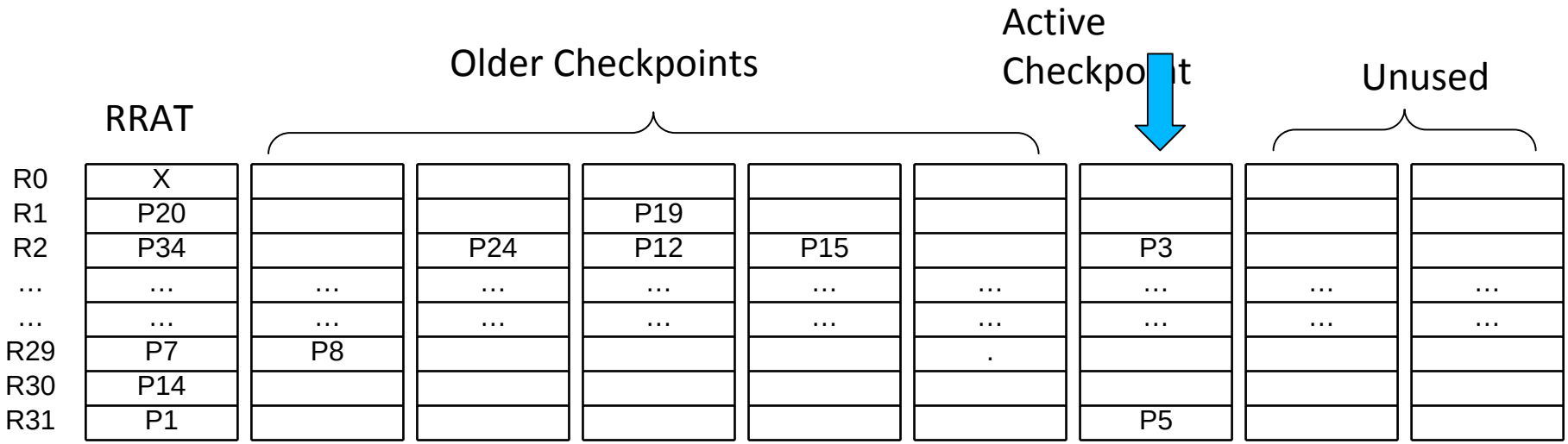
Motivation



Motivation

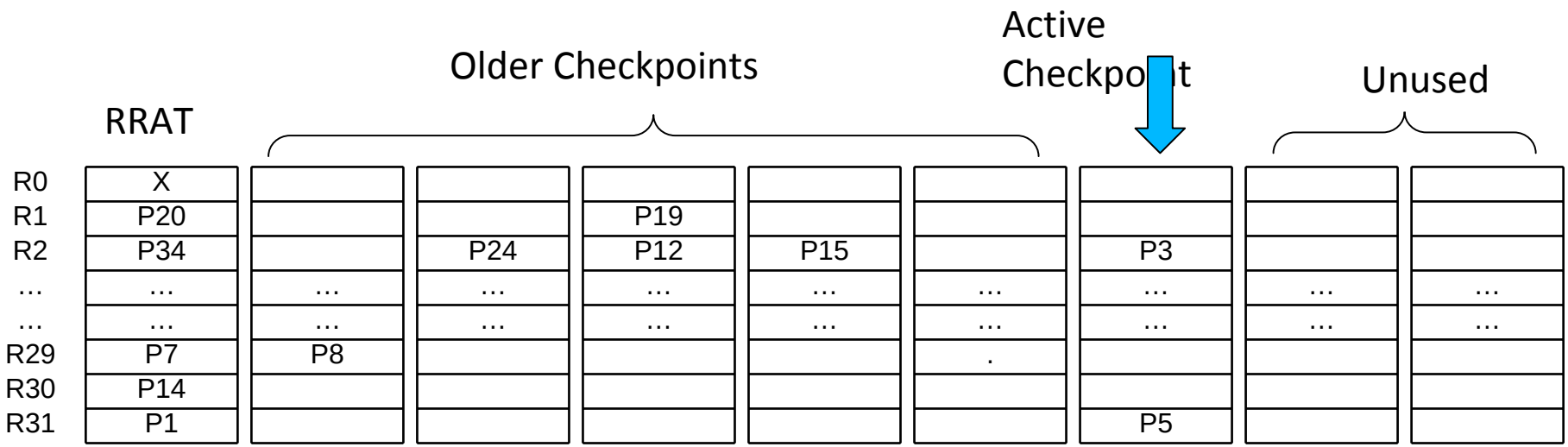
- Hence checkpoint creation and restoration should not involve copying of any large structures on an FPGA
- This is called copy-free checkpointing (CFC)
- The following slides show the design and implementation of CFC

Design Overview



- There are at most 7 *frozen* checkpoint copies + 1 active checkpoint of the RAT
- There is also a committed copy of the RAT (RRAT or Retirement RAT)
- The checkpoints are incremental i.e. only new mappings after taking a checkpoint are added
- For getting the mapping of an architectural register, we use the most recently checkpointed mapping for that register (see figure).
- This is basically a priority encoder

Design Overview



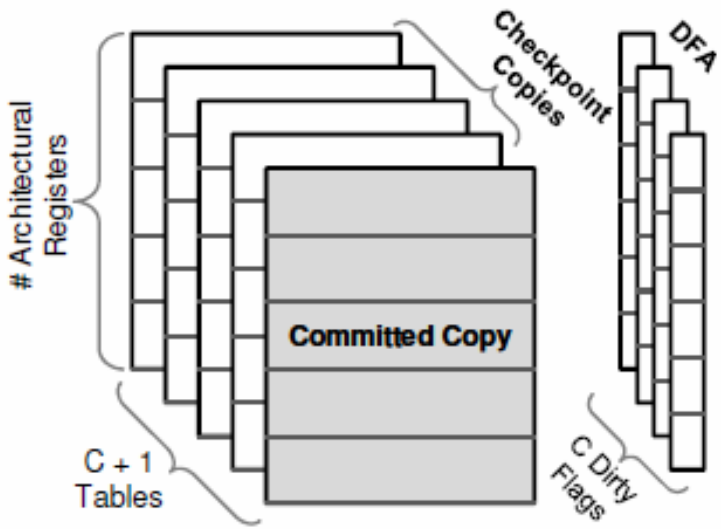
- Some mappings in the above design:
  - R1->P19
  - R2->P3
  - R29->P8
  - R30->P14
  - R31->P5

Design Overview

- A 32-bit Dirty Flag Array (DFA) is kept for each checkpoint
- Each bit in this array reflects whether the corresponding mapping in this checkpoint is valid or not
- Using the DFAs and the RAT, we can determine the most recent mapping for a register (discussed later in detail)

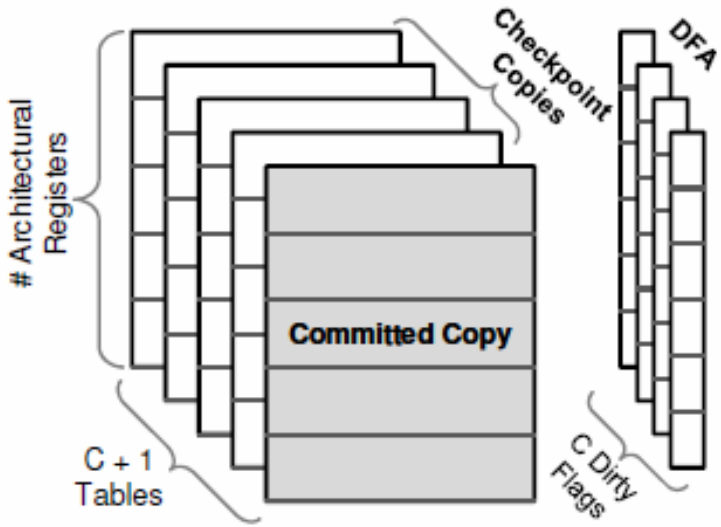
Operation Overview

- Initially, we have the RRAT and an active checkpoint.
- The active checkpoint has no valid entries since new mappings are yet to be created.
- Now, when we start dispatching non-branch, register writing instructions, we rename the destination registers.
- These new mappings are stored in the active checkpoint



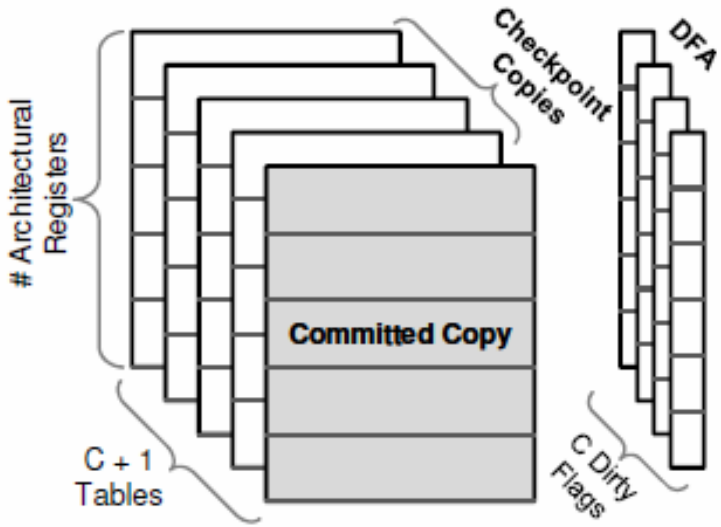
Operation Overview

- Now, lets say a branch gets dispatched.
- We freeze the currently active checkpoint and make no further modifications to it.
- We create a new active checkpoint and any RAT modifications are to be made to this.
- Note that no array copying was involved during checkpointing (we did not copy values from RRAT or the old checkpoint to the newly active one)



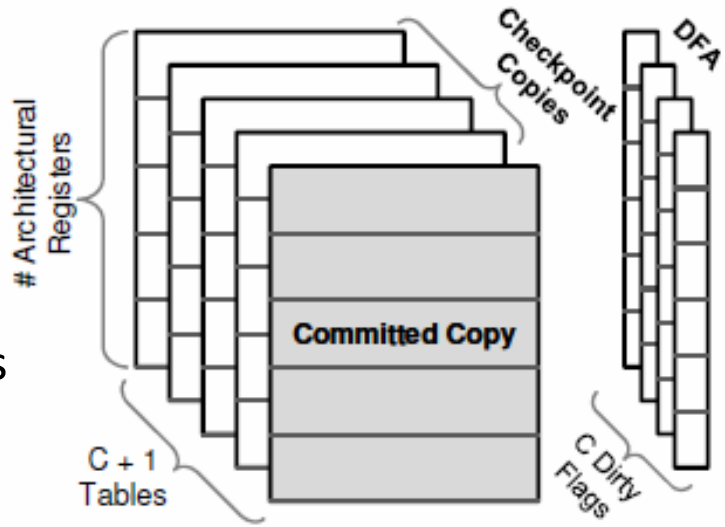
Operation Overview

- When a branch gets committed, we discard the corresponding frozen checkpoint
- Note that it is safe to do so because the RRAT would already have all the correct mappings
- This is because RRAT is updated at every instruction commit
- Note also that the discarded checkpoint is the oldest checkpoint



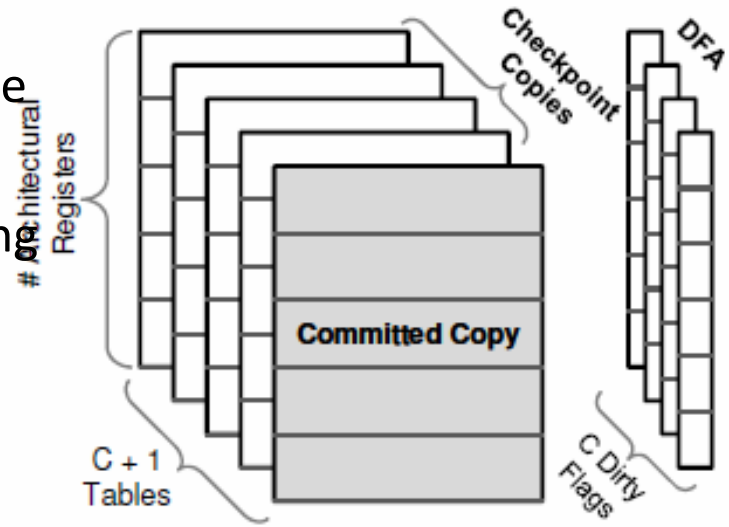
Operation Overview

- When a branch is mispredicted, make the corresponding frozen checkpoint the active one and discard every checkpoint after that
- Of course, we flush all the instructions from ROB and the backend after the branch and start from the new path
- Again, note that no copying is involved!



Pros/Cons

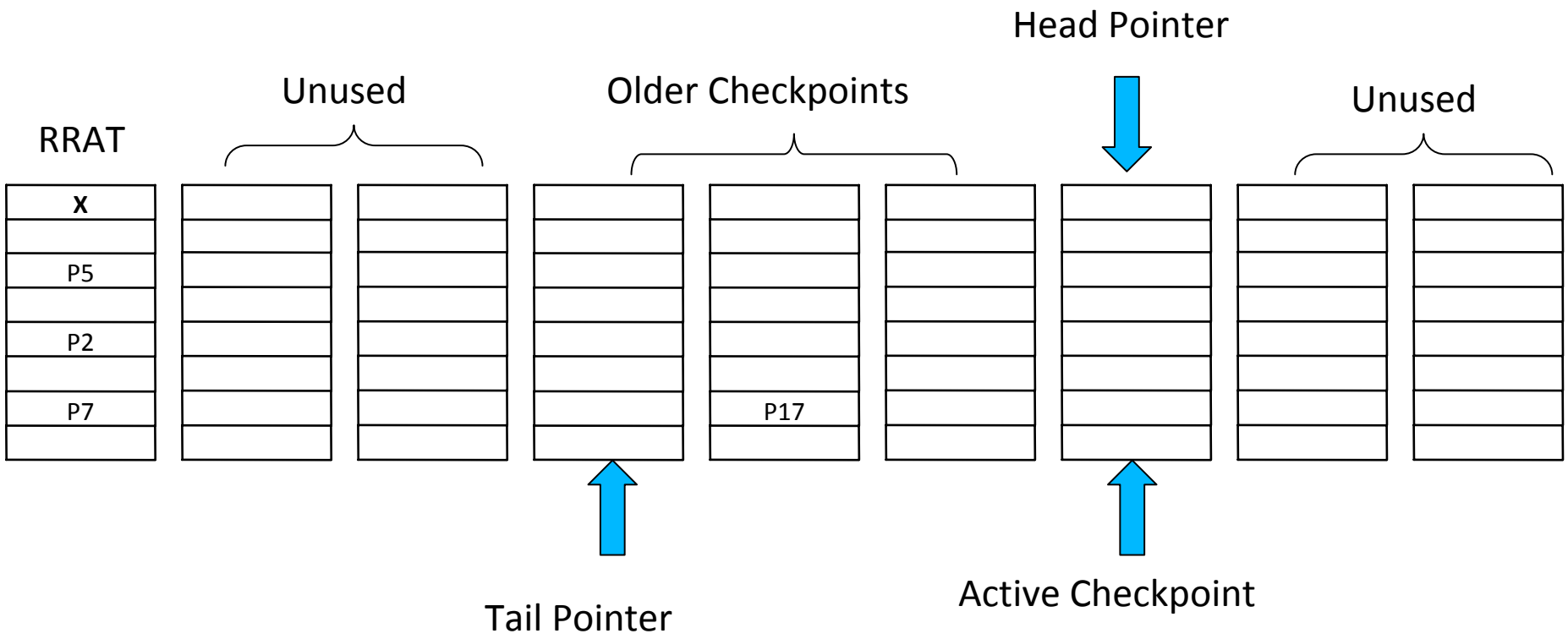
- Copy-free design doesn't come for free
- The penalty of this is a larger clock since while trying to find out a mapping for a register we need to search through the DFAs



Implementation

- 8 checkpoints (upto 7 old + 1 active)
- 32 deep (num of arch registers) and 6-bit wide (48 physical registers)
- Coded as a single 6-bit wide BRAM having 32*8 = 256 entries (Cfc_BRAMarray).
- To access the mapping for register $j from checkpoint id *i,* access Cfc_BRAMarray(i)(j)
- The list of checkpoints is accessed as an 8-entry FIFO
- There are two **3 bit** internal pointers, head pointer & tail pointer pointing to the active checkpoint and the oldest non-committed branch respectively
  - Note: one checkpoint is always used as the active one, and hence only 3 bits. The full condition is that head-tail = 7.
- During Reset, both are initialized to zero (000)
- The head pointer is incremented when a new branch is dispatched while the tail pointer is incremented when a branch commits from top of the ROB

Implementation



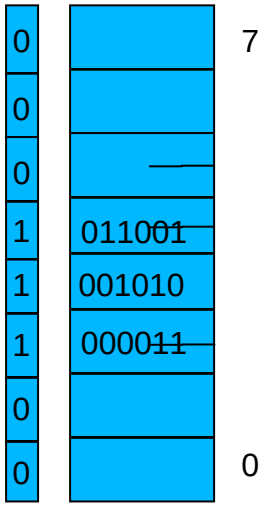
Implementation

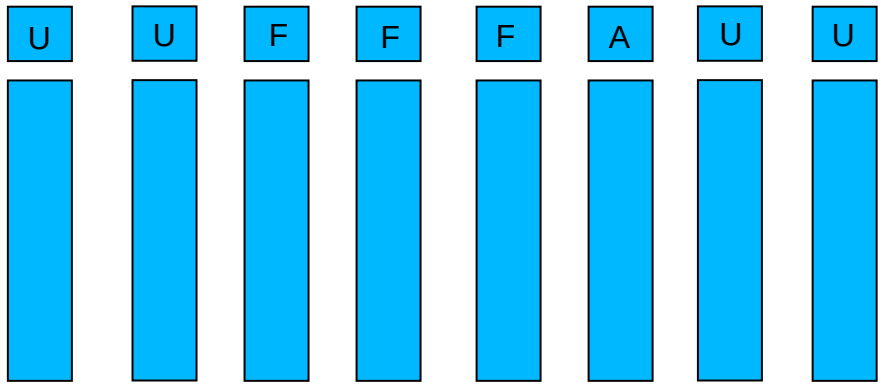
- BRAMs are dual-ported (1 read and 1 write)
- During dispatch, we need Rs, Rt and Rd architectural -> physical register mappings
- Hence we need 3 read ports!
- So we make 3 copies of the above BRAM
- Writing done to active checkpoint during dispatch (when Rd register is remapped)
- Only one write port is needed
- During writing, we write the same data to each of the 3 copies

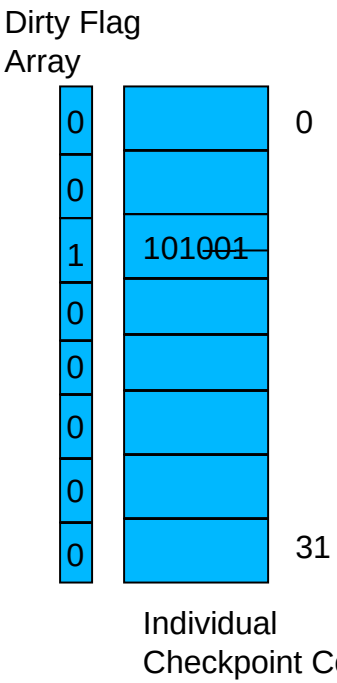
Implementation

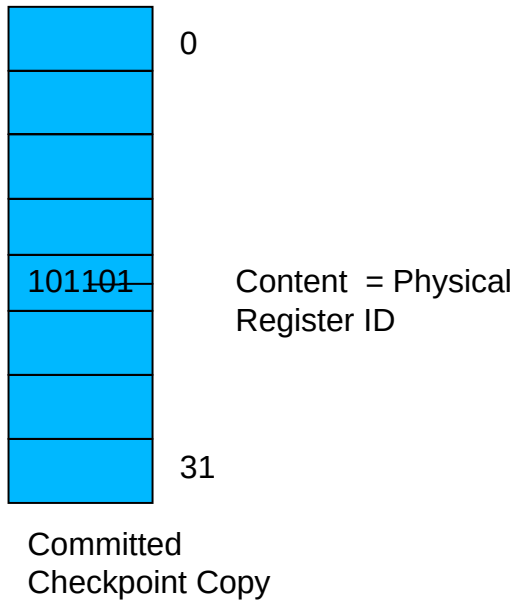
- There is also one 32 deep and 6 bit wide committed checkpoint copy (RRAT).
- When an R-type or load word instruction commits, the physical register mapping for the corresponding Rd register for that instruction is now final
- This mapping is written to the RRAT
- Dirty Flag Array (DFA)
    - This is needed to indicate which mappings in a checkpoint are valid. Hence 32 deep single bit wide array are needed per checkpoint. Corresponding to 8 checkpoint copies, 8 such arrays are needed.
- ROBTags of upto 7 branches are also stored in a Checkpoint Tag Array, corresponding to 7 frozen checkpoints at max.

Note: At a time only 7 checkpoints can be frozen, as 1 checkpoint always needs to be Active


0
0
0
1
1
0
0
011001
001010
000011
7
0
Head_Pointer
Content =
ROB Tag
Tail_Pointer
Checkpoint
Valid Array
Checkpoint
Tag Array


8 Checkpoint Copies
A – Active Copy
F – Frozen Copy
U – Unused Copy
U
U
F
F
F
A
U
U


Dirty Flag
Array
0
0
1
0
0
0
0
0
101001
0
31
Content = Physical
Register ID
Individual
Checkpoint Copy


101101
0
31
Content = Physical
Register ID
Committed
Checkpoint Copy

Functioning

- The code can be divided into three parts:

Tasks to be done when a new non branch instruction is dispatched

Tasks to be done when a new non branch instruction is committed

Tasks to be done when new branch instruction is dispatched

Tasks to be done when new branch instruction is committed

Tasks to be done when the branch is mispredicted

When a non-branch instruction is Dispatched

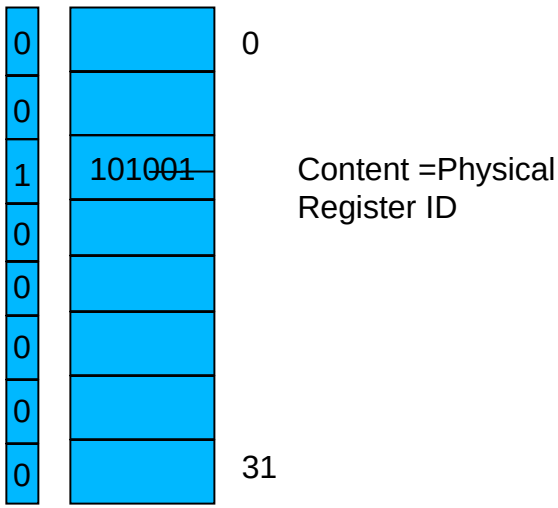
- When a new register writing instruction is being dispatched, the corresponding new physical mapping for Rd is stored in the active checkpoint at Rd's location

- The 8 bit address for accessing the BRAM is obtained by concatenating the 3 bit head pointer value with the 5 bit logical Rd value

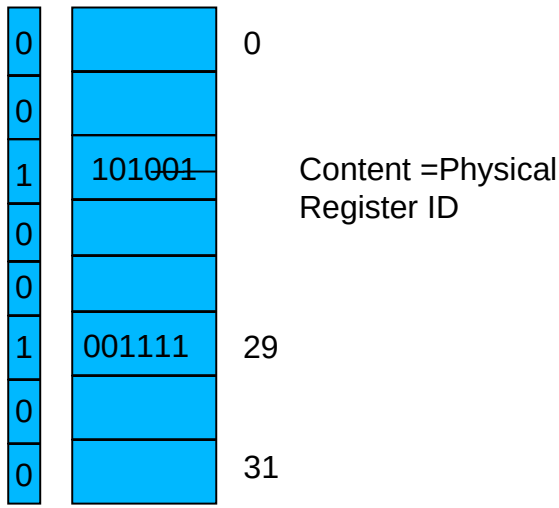
- At the same time, the DFA bit in the active checkpoint corresponding to Rd location is set to 1, indicating the value at that location is valid

When a non-branch Instruction writing to Architectural Register 29 is dispatched with corresponding mapping to Physical Register 15

Before



After



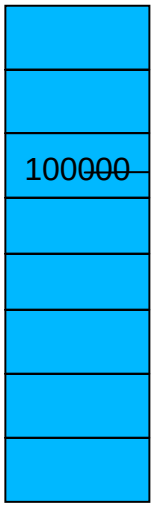
When a non-branch instruction is Committed

When Register-Writing instruction is committed, the committed RAT is changed to write the Rd register mapping.

- The old physical register mapped to Rd would be freed and written to FRL
- There is no need to change the DFA bits corresponding to that Rd register.

When a non-branch Instruction writing to Architectural Register 23 & mapped to Physical Register 45 is committed from Rob Top

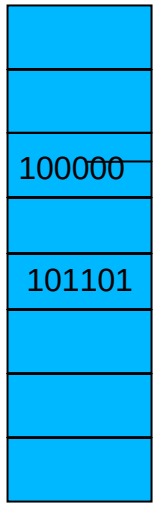
Before



Content = Physical Register ID

Committed Checkpoint Copy

After



Content = Physical Register ID

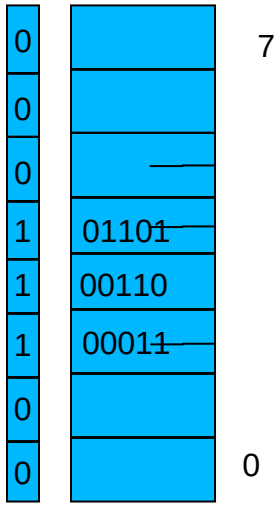
Committed Checkpoint Copy

When a new branch instruction is dispatched

- Whenever a new branch instruction is dispatched, the current active checkpoint is frozen and the value of head pointer is incremented by 1
- The corresponding ROB tag of the new branch instruction is stored in the CFC Tag Array
- If all the checkpoints are occupied (i.e. All 7 checkpoints are frozen) and a branch instruction comes at dispatch, then we stall the dispatch until the seniormost branch rises to the top of ROB and releases its checkpoint.

When a New Branch Instruction is Dispatched with ROB Tag 25

Before



Checkpoint Valid Array

Checkpoint Tag Array

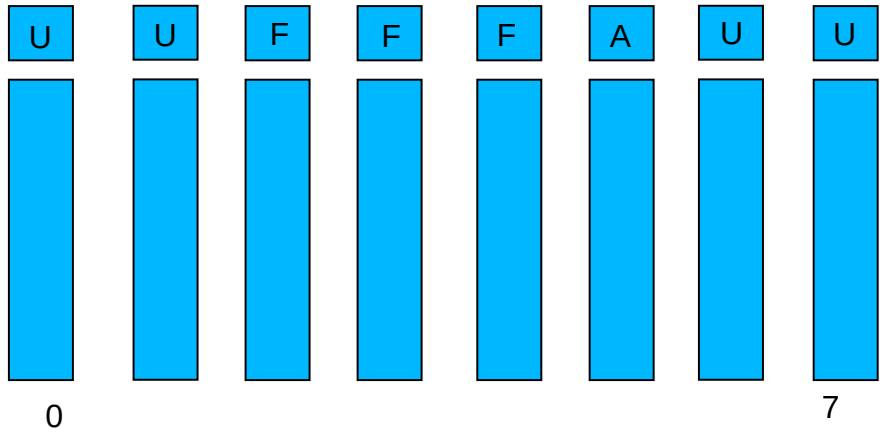
Head_Pointer

Content = ROB Tag

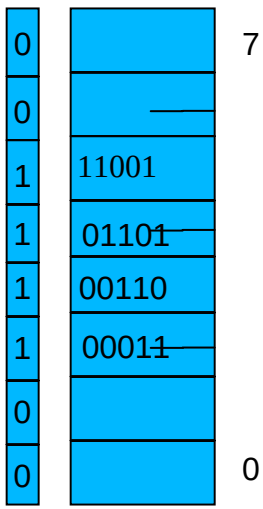
Tail_Pointer

A – Active Copy
F – Frozen Copy
U – Unused Copy

8 Checkpoint Copies



After



Checkpoint Valid Array

Checkpoint Tag Array

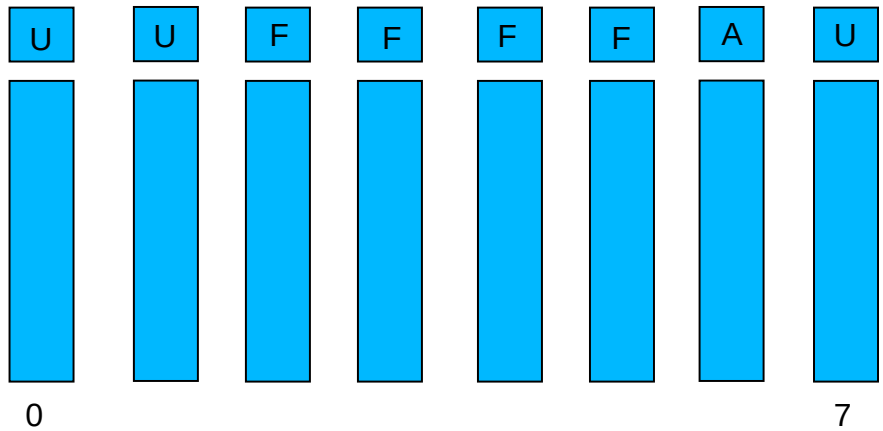
Head_Pointer

Content =ROB Tag

Tail_Pointer

A – Active Copy
F – Frozen Copy
U – Unused Copy

8 Checkpoint Copies

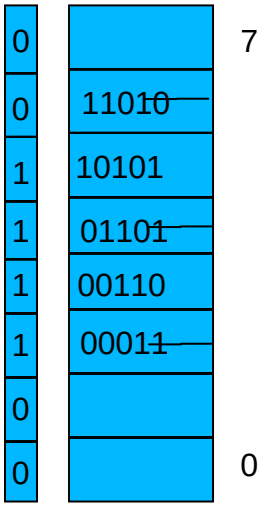


When a Branch Instruction Commits from the Top of the ROB

- Whenever a branch instruction commits from the top of the ROB, the checkpoint pointed by the tail pointer is released and the value of tail pointer is increased by 1.
- At the same time, all the DFA bits of the Dirty Flag Array corresponding to that checkpoints are reset to 0.

When a Branch Instruction Commits from the top of the ROB

Before



Checkpoint Valid Array

Checkpoint Tag Array

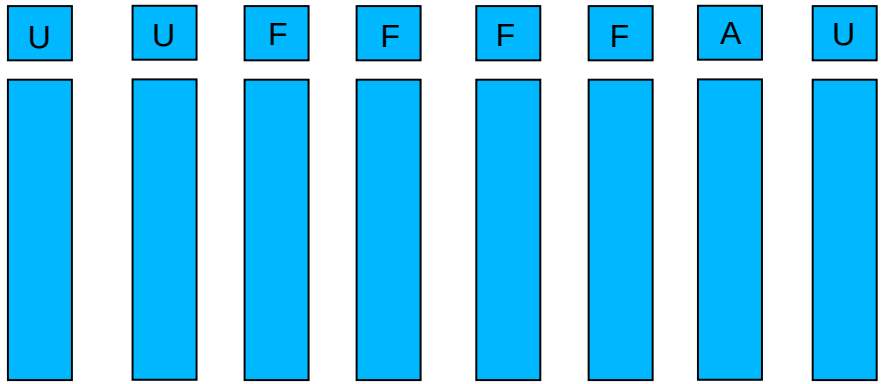
Head_Pointer

Content = ROB Tag

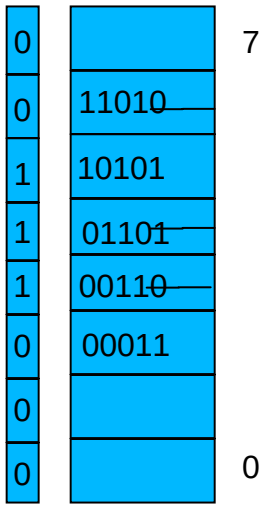
Tail_Pointer

A – Active Copy
F – Frozen Copy
U – Unused Copy

8 Checkpoint Copies



After



Checkpoint Valid Array

Checkpoint Tag Array

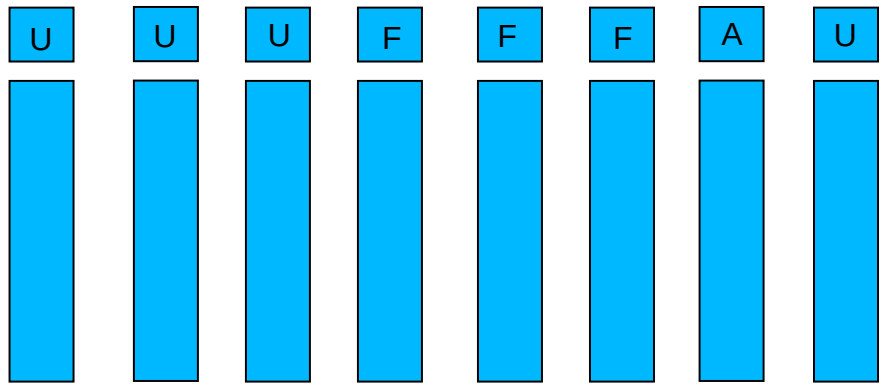
Head_Pointer

Content - ROB Tag

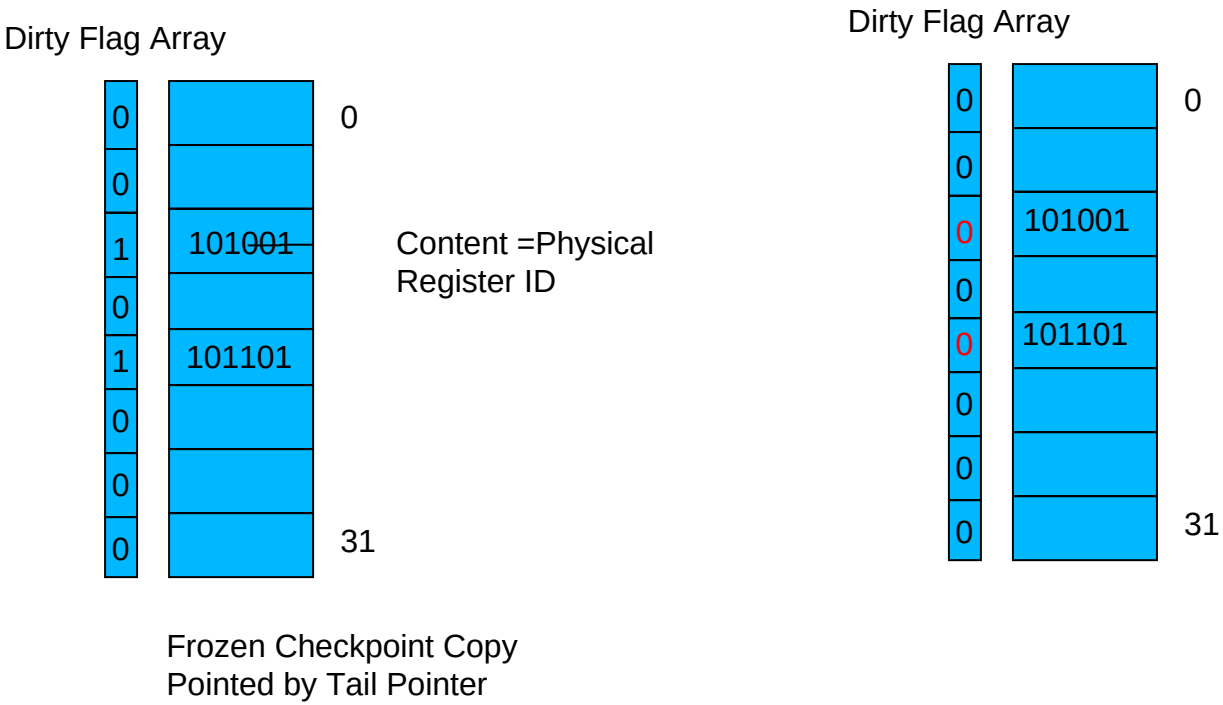
Tail_Pointer

A – Active Copy
F – Frozen Copy
U – Unused Copy

8 Checkpoint Copies



Corresponding change in the Freed Copy

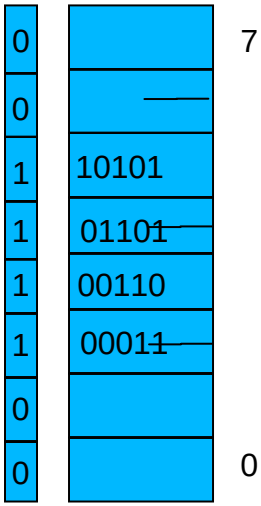


When Branch is Mis-predicted

- Whenever a branch is mis-predicted, the head pointer is simply moved to the checkpoint table created by the corresponding branch freeing 1 or more checkpoints (the current active checkpoint plus any other)
- The checkpoint corresponding to that branch becomes the new active checkpoint
- At the same time, all the DFA bits of the freed checkpoints are reset to 0

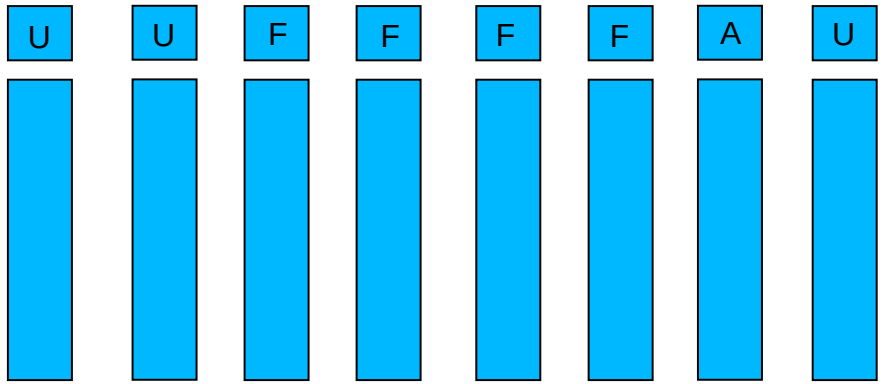
When a Branch Instruction with ROB Tag 13 gets Mispredicted

Before

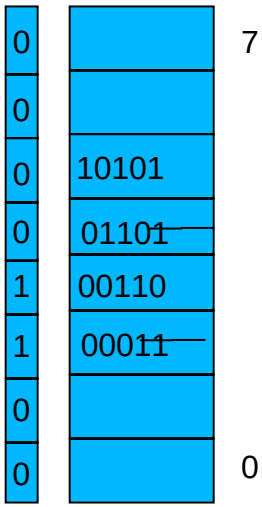


A – Active Copy
F – Frozen Copy
U – Unused Copy

8 Checkpoint Copies

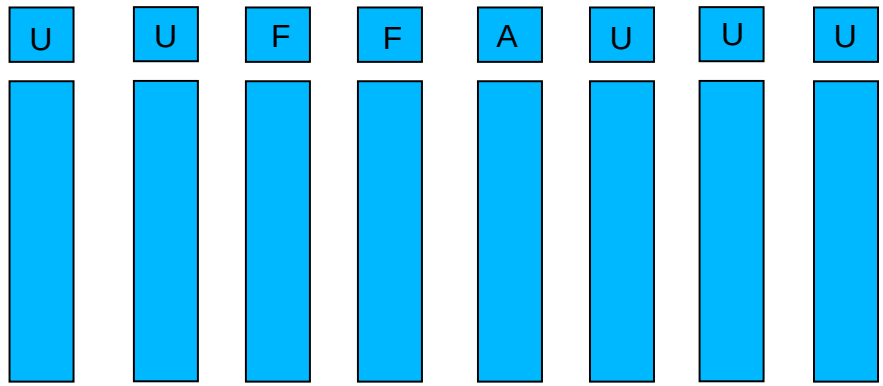


After



A – Active Copy
F – Frozen Copy
U – Unused Copy

8 Checkpoint Copies

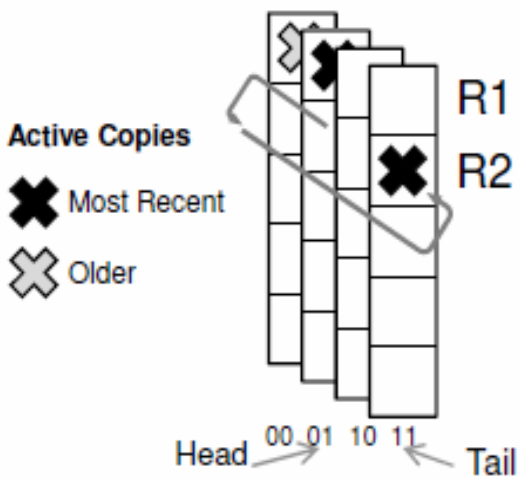


Expanded on Next Page


According changes in the Dirty Flag Array Copies corresponding to the Individual Checkpoint Copies
Dirty Flag Array
0
0
0
1
0
1
0
0
000011
000110
0
31
Before
Frozen Checkpoint with
ROB Tag 13
Dirty Flag Array
0
1
0
0
0
0
0
0
001000
0
31
Frozen Checkpoint with
ROB Tag 21
Dirty Flag Array
0
0
1
0
1
0
0
0
101001
101101
0
31
Content Physical
Register Mapping
Active Checkpoint Copy
Pointed by Head Pointer
Dirty Flag Array
0
0
0
1
0
1
0
0
000011
000110
0
31
After
Active checkpoint
Dirty Flag Array
0
0
0
0
0
0
0
0
001000
0
31
Unused
checkpoint
Dirty Flag Array
0
0
0
0
0
0
0
0
101001
101101
0
31
Content Physical
Register Mapping
Unused checkpoint

Finding the latest mapping for a Register

- Finding the table holding the latest value for an architectural register (rs, rd & rt) is achieved by **searching through the DFA row of that register, i.e.** search among the 8 checkpoints which one contains the **most recent valid data**. It is essentially finding the first dirty flag set moving from head pointer to tail pointer. If none is set, then the latest value is in the committed table.
- Since this is a circular FIFO and head pointer's numeric value may be both greater or less than the tail pointer's value, searching is a little involved.
- For register $20, for instance, we would search the row DFA_array(i)(20) where i ranges from 0 to 7.



Finding the latest mapping for a Register

- The scheme used to do the search is as follows:
  - At first, the DFA row from Head_Pointer to index 0 is searched. If any DFA bit is found to be set, than we take corresponding value from that checkpoint.
  - If not, than the DFA row from 7 to Tail_Pointer is searched and if found any DFA bit to be set, the corresponding value is taken.
  - If DFA bit is not found to be set in any of the above two cases, then the value is taken from the Committed Checkpoint.
- Notice that the DFA bits outside of the search region are always set to 0

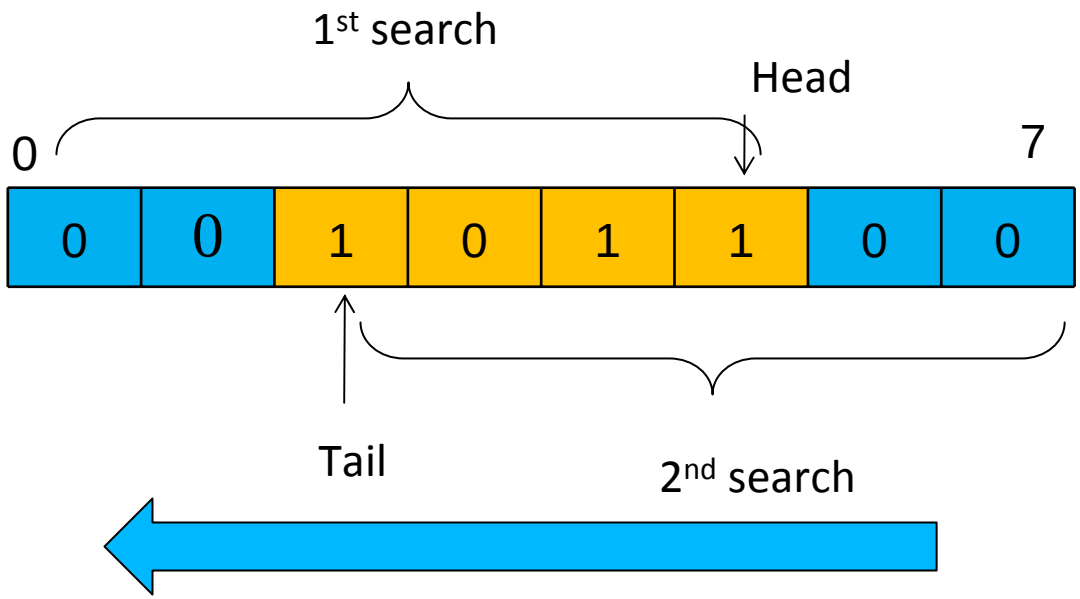
DFA Search Process

Case 1: head > tail

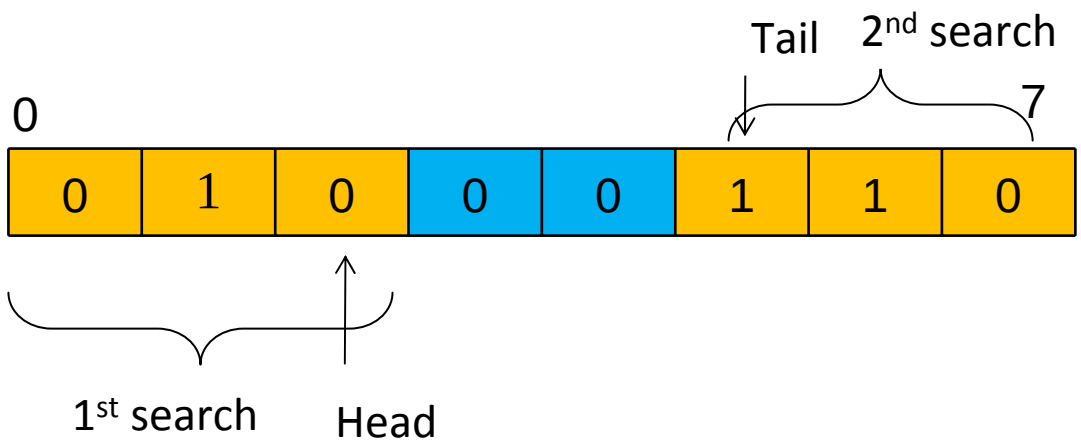
Area To be Searched

Invalid locations

Note that the Search Process remains the same in both cases; these cases have been shown here for your understanding. In Case 1, the 1st and the 2nd search would always yield the same answer. Two searches are needed because of Case 2.



Search Direction: Right to Left in all cases



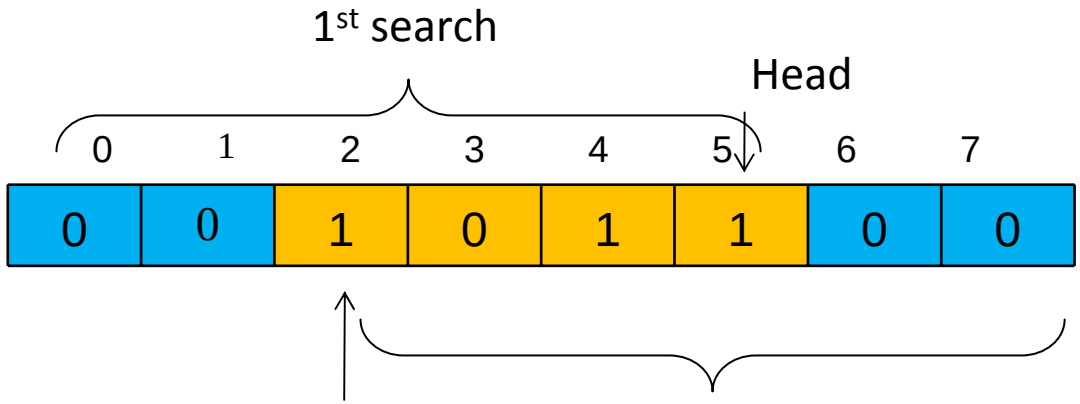
Case 2: head < tail

DFA Search Example 1

Case 1: head > tail

Area To be Searched

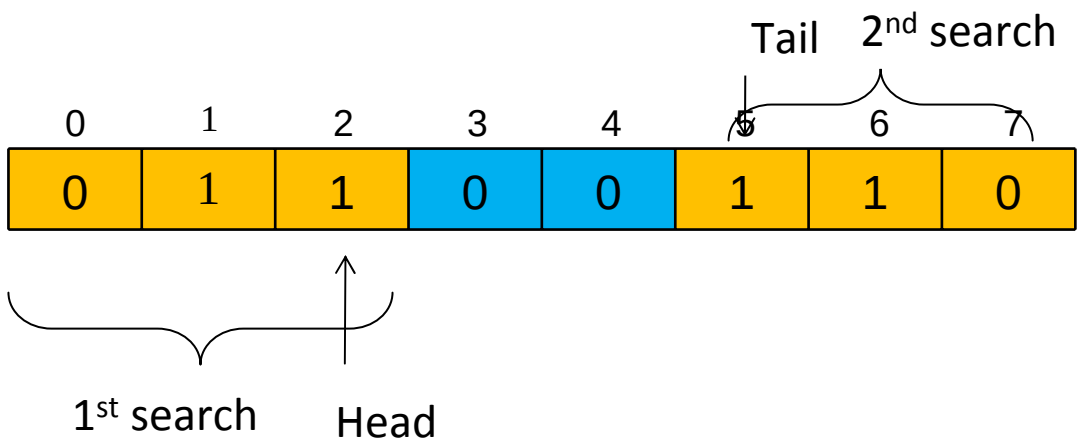
Invalid locations



In Case 1: checkpoint #5 (active checkpoint) contains the most recent mapping

In Case 2: 1st search finds that checkpoint #1 (active checkpoint) contains the most recent mapping.

Search Direction: Right to Left in all cases



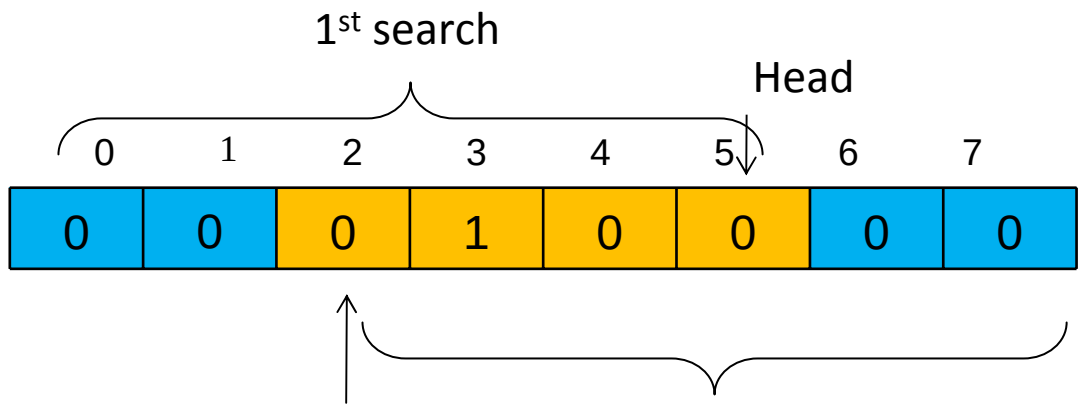
Case 2: head < tail

DFA Search Example 2

Case 1: head > tail

Area To be Searched

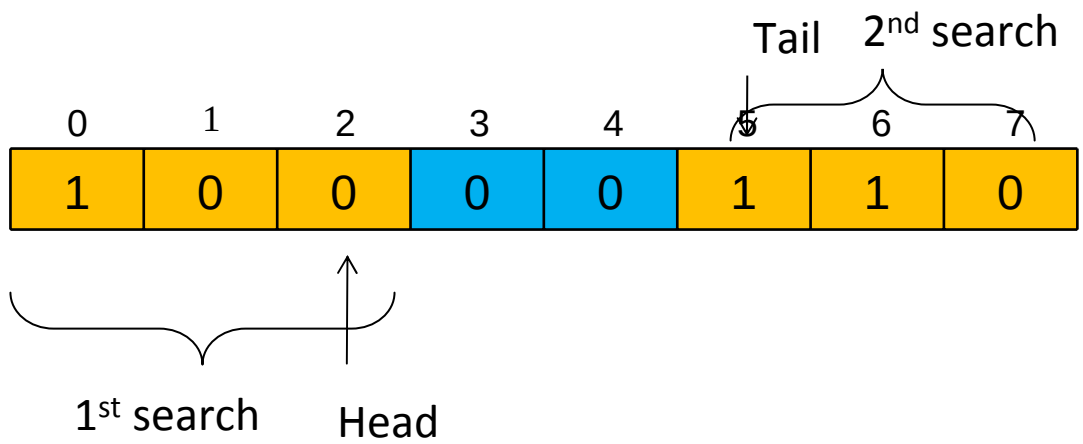
Invalid locations



In Case 1: checkpoint #3 contains the most recent mapping

In Case 2: 1st search finds that checkpoint #0 contains the most recent mapping.

Search Direction: Right to Left in all cases



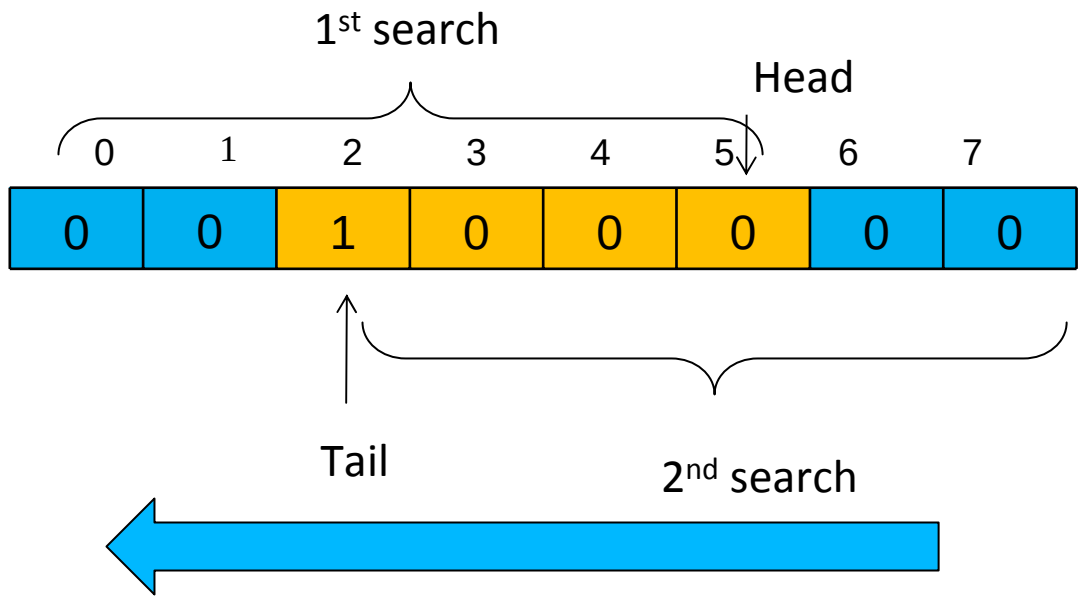
Case 2: head < tail

DFA Search Example 3

Case 1: head > tail

Area To be Searched

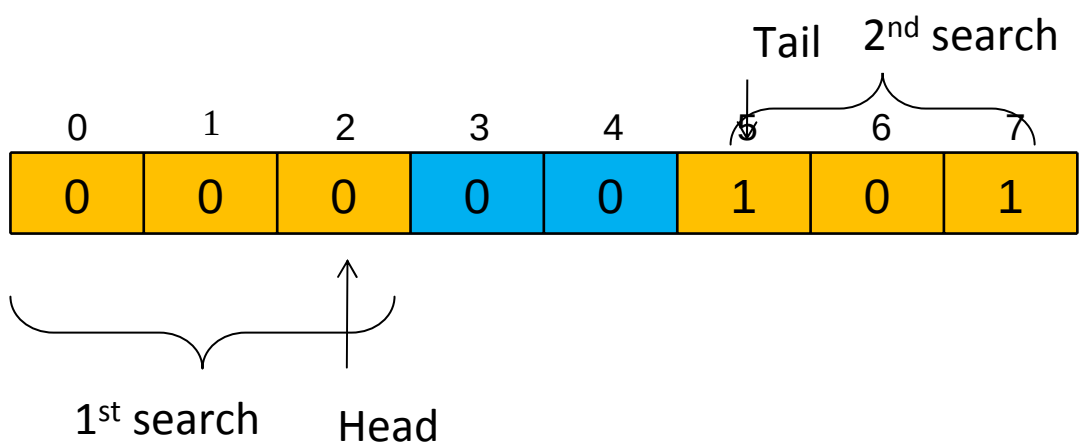
Invalid locations



In Case 1: checkpoint #3 contains the most recent mapping

In Case 2: 1st search does not find any valid locations. 2nd search finds that checkpoint #7 is the most recent mapping.

Search Direction: Right to Left in all cases



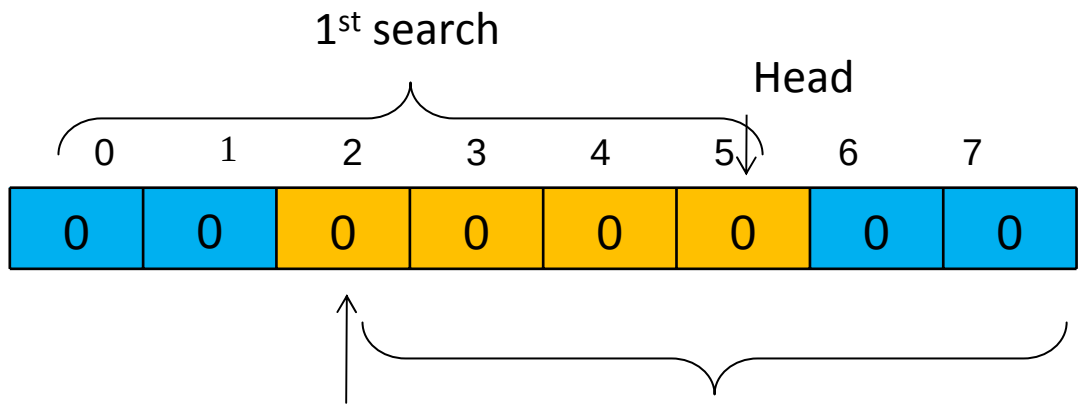
Case 2: head < tail

DFA Search Example 4

Case 1: head > tail

Area To be Searched

Invalid locations



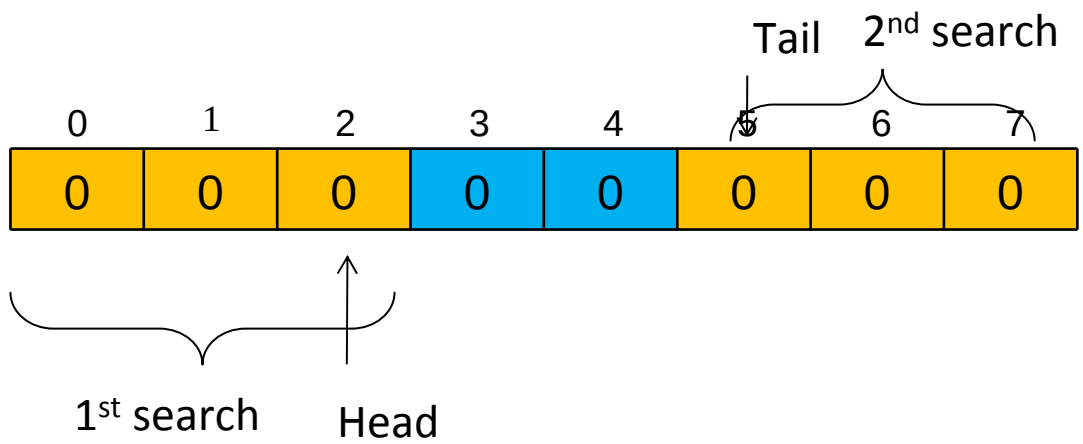
Tail

2nd search

In Case 1: Both searches do not find a valid mapping in the checkpoints. Hence committed mapping is used.

In Case 2: Same as above

Search Direction: Right to Left in all cases



Case 2: head < tail

Interface with FRL

- The Free Register List (FRL)'s state should also be saved/rolled back when a branch is dispatched/mispredicted.
- This can be done simply by saving the FRL's head pointer during dispatch of a branch and restoring it back when that branch is mispredicted (see FRL's slides for detail)
- The CFC module stores FRL head pointers corresponding to each branch it checkpoints. This is an array of 8 locations (but only 7 are used) each 5-bit wide (16-location FRL FIFO)
- CFC gets the FRL head pointer's value from the FRL module during dispatch of a branch
- When a branch is mispredicted (CDBFlush Is true), it gives the FRL headpointer corresponding to the mispredicted branch to the FRL