Course on: "Advanced Computer Architectures"

INSTRUCTION LEVEL PARALLELISM: REORDER BUFFER

Prof. Cristina Silvano
Politecnico di Milano
cristina.silvano@polimi.it

Outline

Hardware-based Speculation

Reorder Buffer

Speculative Tomasulo Algorithm

Hardware-based Speculation

Branches must be solved quickly for loop overlap!

- In our loop-unrolling example, we relied on the fact that branches were under control of “fast” integer unit in order to get overlap!

```
Loop:   LD      F0   0    R1
        MULTD   F4   F0   F2
        SD      F4   0    R1
        SUBI    R1   R1   #8
        BNEZ    R1   Loop
```

- What happens if branch depends on result of **MULTD**??
  - We completely lose all of our advantages!
  - **Need to be able to “predict” the branch outcome.**
  - If we were to predict that branch was **taken**, this would be right most of the time.
- Problem **much** worse for superscalar machines!

Hardware-based Speculation

- The key idea behind speculation is:
  - to issue and execute instructions dependent on a branch **before** the branch outcome is known;
  - to allow instructions to **execute out-of-order** but to force them to **commit in-order;**
  - to prevent any irrevocable action (such as updating state or taking an exception) until an instruction commits;
- When an instruction is no longer speculative, we allow it to update the register file or memory ***(instruction commit).***
- ***ReOrder Buffer (ROB)*** *to hold the results of instructions that have completed execution but have not yet committed or to pass results among instructions that may be speculated.*

Hardware-based Speculation

- Outcome of branches is **speculated** and program is **executed as if speculation was correct** (without speculation, the simple dynamic scheduling would only fetch and decode, *not execute!)*
- Mechanisms are necessary to handle incorrect speculation – **hardware speculation** extend dynamic scheduling beyond a branch (i.e. behind the basic block)

HW-based Speculation

- **HW-based Speculation** combines 3 ideas:
  - **Dynamic Branch Prediction** to choose which instruction to execute before the branch outcome is known
  - **Speculation** to execute instructions before control dependences are resolved
  - **Dynamic Scheduling** supporting out-of-order execution but **in-order commit** to prevent any irrevocable actions (such as register update or taking exception) until an instruction commits

Hardware-based Speculation

- Adopted in PowerPC 603/604, MIPS R10000/ R12000, Pentium II/III/4, AMD K5/K6 Athlon.
- Extends hardware support for Tomasulo algorithm: to support speculation, ***commit*** phase is separated from execution phase, ***ReOrder Buffer*** is introduced → ***Speculative Tomasulo Algorithm with ReOrder Buffer***

Hardware-based Speculation

- Basic Tomasulo algorithm: instruction writes result in RF and Reservation Stations, where subsequent instructions find the results to be used as operands
- With speculation, results are written only when instruction ***commits*** – and it is known whether the instruction (no more speculative) had to be executed.
- ***Key idea: executing out of order, committing in order.***

ReOrder Buffer (ROB)

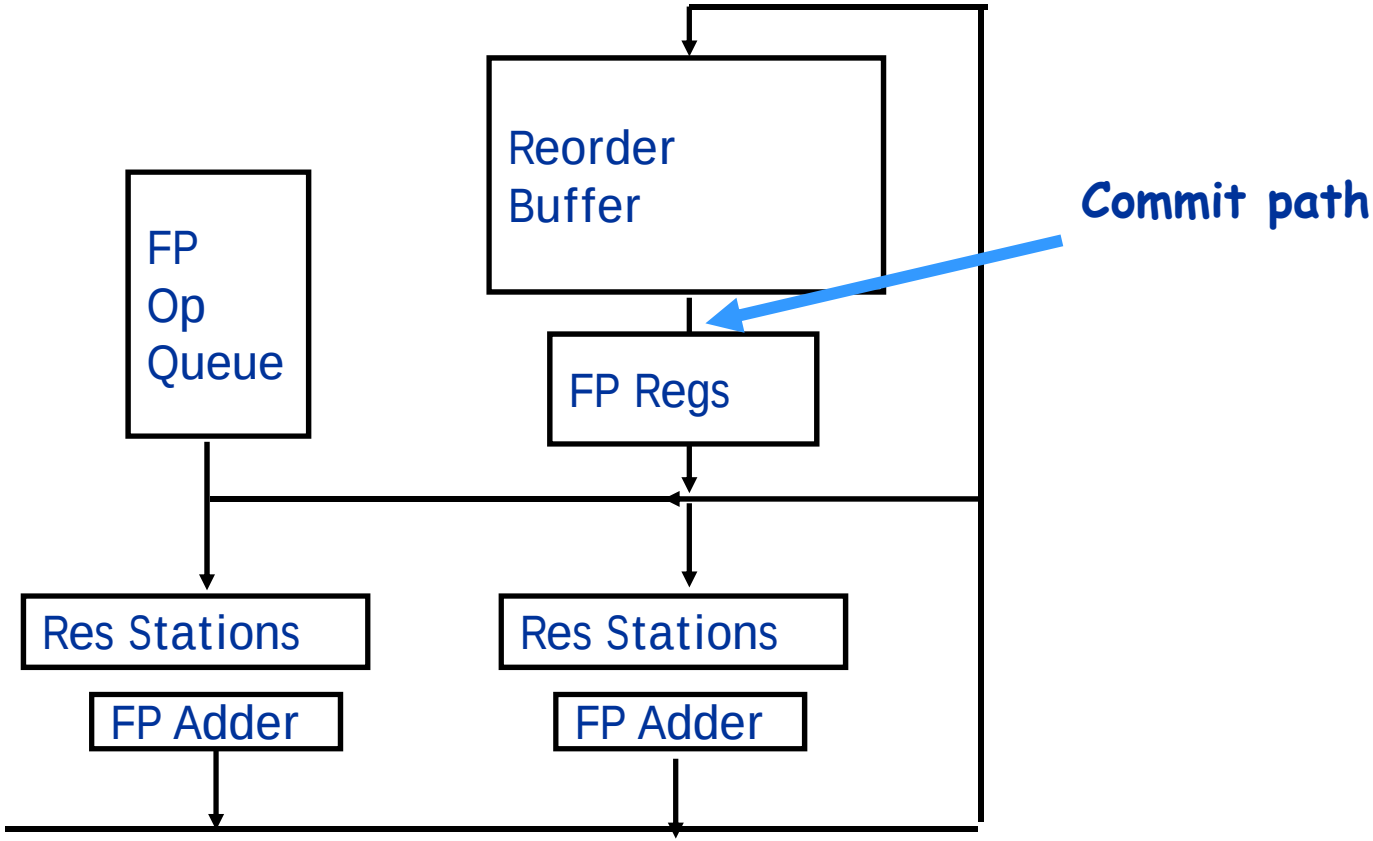
ReOrder Buffer (ROB)

- **Buffer to hold the results of instructions that have finished execution but non committed**
- Buffer to pass results among instructions that can be speculated
- **Support *out-of-order* execution but *in-order* commit**

ReOrder Buffer (ROB)

- **Buffer to hold the results of instructions that have finished execution but non committed**
- Buffer to pass results among instructions that can be speculated
- **Support *out-of-order* execution but *in-order* commit**
- **Speculative Tomasulo Algorithm with ROB:**
  - Pointers are directed toward ROB slot.
  - A register or memory is updated only when the instruction reaches the head of ROB (that is until the instruction is no longer speculative).

ReOrder Buffer (ROB)



ReOrder Buffer (ROB)

- **ROB completely replaces the Store Buffers**
- **The renaming function of Reservation Stations is replaced by ROB**
- Reservation Stations now used only to buffer instructions and operands to FUs (to reduce structural hazards).
- **Pointers now are directed toward ROB entries**
- Processors with ROB can dynamically execute while maintaining precise interrupt model because instruction commit happens in order.

ReOrder Buffer (ROB)

- **Reorder buffer can be operand source ⇒ More registers like reservation stations**
- Use reorder buffer number instead of reservation station when execution completes
- Supplies operands between execution complete & commit
- Once operand commits, result is put into register
- Instructions **commit**
- As a result, its easy to undo speculated instructions on mispredicted branches **or on exceptions**

ReOrder Buffer (ROB)

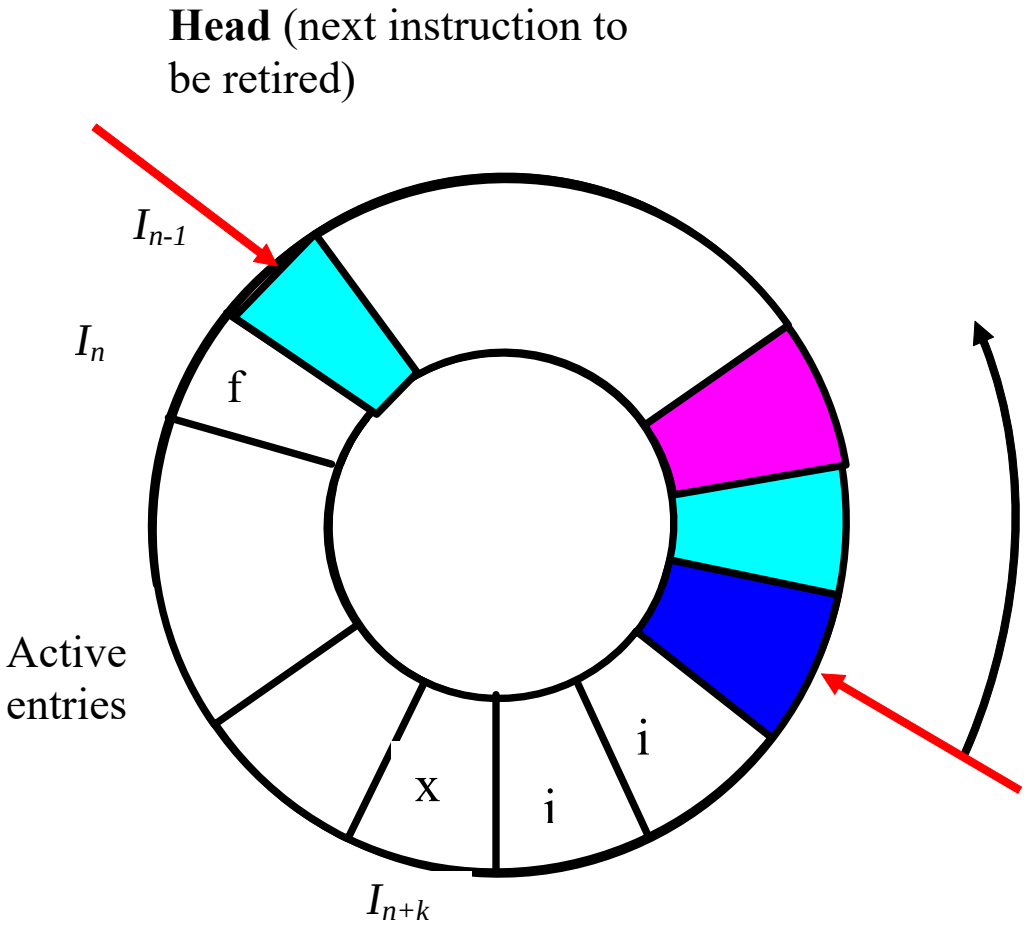
- Originally (1988) introduced to solve precise interrupt problem; generalized to grant sequential consistency;
- Basically, ROB is a **circular buffer** with **tail pointer** (indicating next free entry) and **head pointer** indicating the instruction that will **commit** (leaving ROB) **first**.

ReOrder Buffer (ROB)

- Instructions are written in ROB in strict program order – when an instruction is issued, an entry is allocated to it **in sequence.** Entry indicates status of instruction: issued (**i**), in execution (**x**), finished (**f**) (**+ other items!**),

- An instruction can ***commit (retire)*** iff
  1. It has finished, and
  2. All previous instructions have already retired.

ReOrder Buffer (ROB)

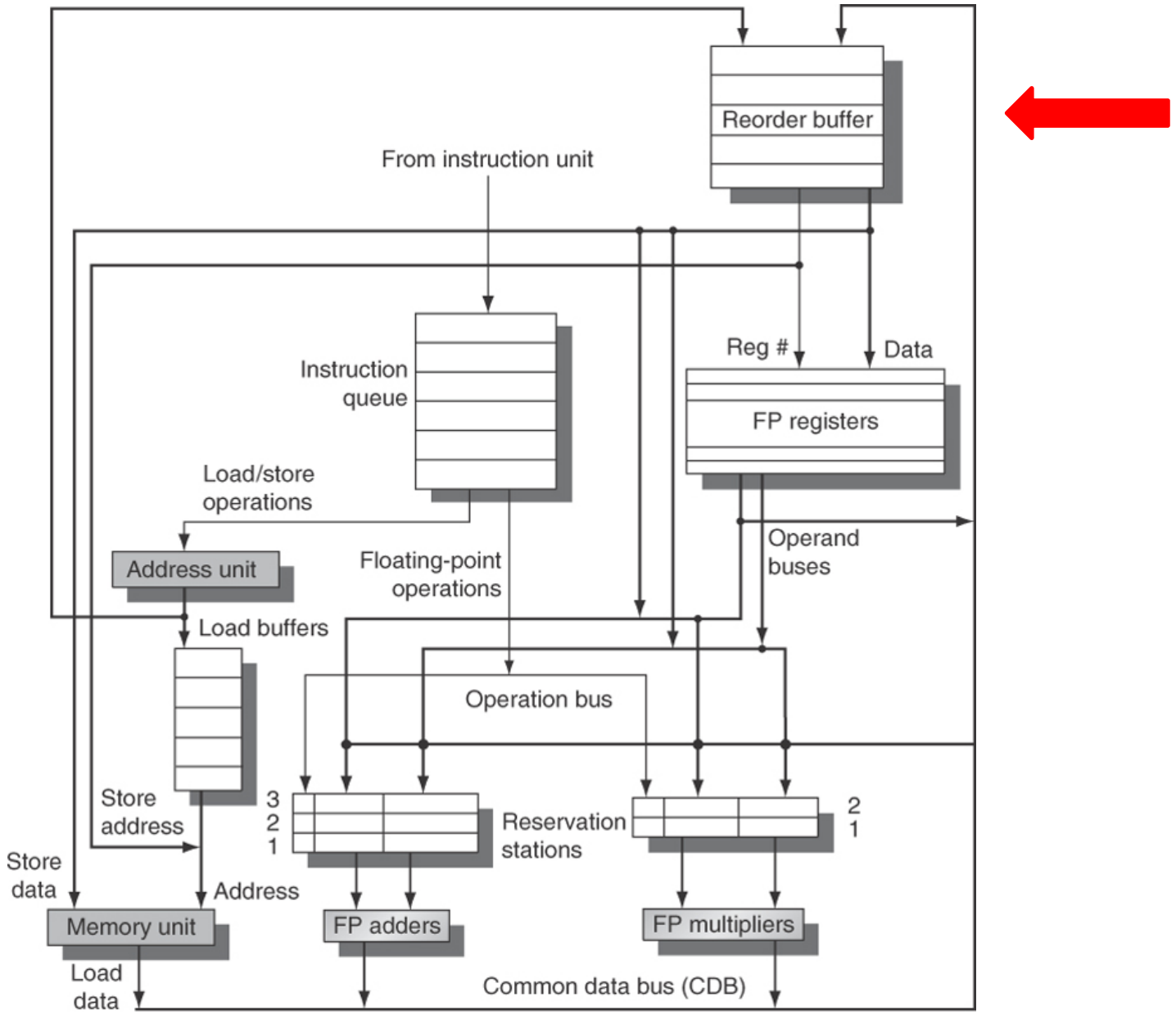


ReOrder Buffer (ROB)

- Only retiring instructions can ***complete***, i.e., **update architectural registers and memory;**
- ROB can support ***both speculative execution and exception handling***
- **Speculative execution**: each ROB entry is extended to include a ***speculative status field***, indicating whether instruction has been executed speculatively;
- Finished instructions ***cannot retire as long as they are in speculative status***.
- **Interrupt handling**: Exceptions generated in connection with instruction execution are made ***precise*** by accepting exception request only when instruction becomes “next to retire” (exceptions are processed *in order*)

Speculative Tomasulo Algorithm

Speculative Tomasulo Architecture for an FPU



Four Steps of Speculative Tomasulo Algorithm

1. **Issue** — get instruction from FP Op Queue
   If reservation station **and ROB slot** free, issue instr & send operands **& ROB no. for destination**
   (this stage sometimes called “dispatch”)
2. **Execution** — operate on operands (EX)
   When both operands ready then execute; if not ready, watch CDB for result; when both operands in reservation station, execute; checks RAW (sometimes called “issue”)
3. **Write result** — finish execution (WB)
   Write on Common Data Bus to all awaiting FUs **& ROB;** mark reservation station available.
4. **Commit — update register with ROB result**
   When instr. at head of ROB & result present, update register with result (or store to memory) and remove instr from ROB. Mispredicted branch flushes ROB (sometimes called “graduation”)

Steps of Speculative Tomasulo's Algorithm (2)

4. **Commit**: 3 different possible sequences:
   1. **Normal commit:** instruction reaches the head of the ROB, result is present in the buffer. Result is stored in the register, instruction is removed from ROB;
   2. **Store commit:** as above, but memory rather than register is updated;
   3. **Instruction is a branch with incorrect prediction:** it indicates that speculation was wrong. ROB is flushed ("graduation"), execution restarts at correct successor of the branch.
      If the branch was correctly predicted, branch is finished

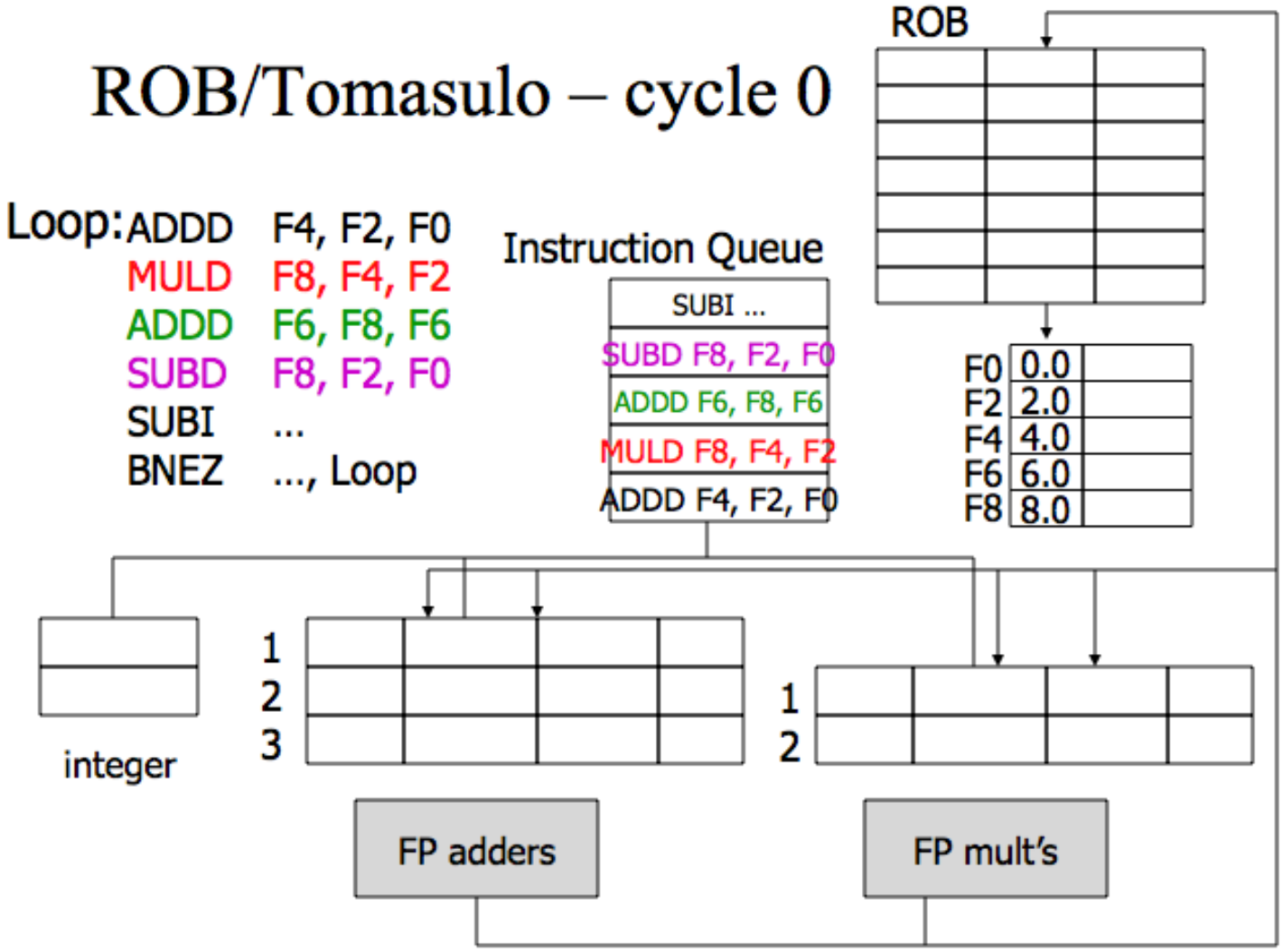
Speculative Tomasulo's Algorithm

- Tomasulo's "Boosting" needs a buffer for uncommitted results **(ROB).**

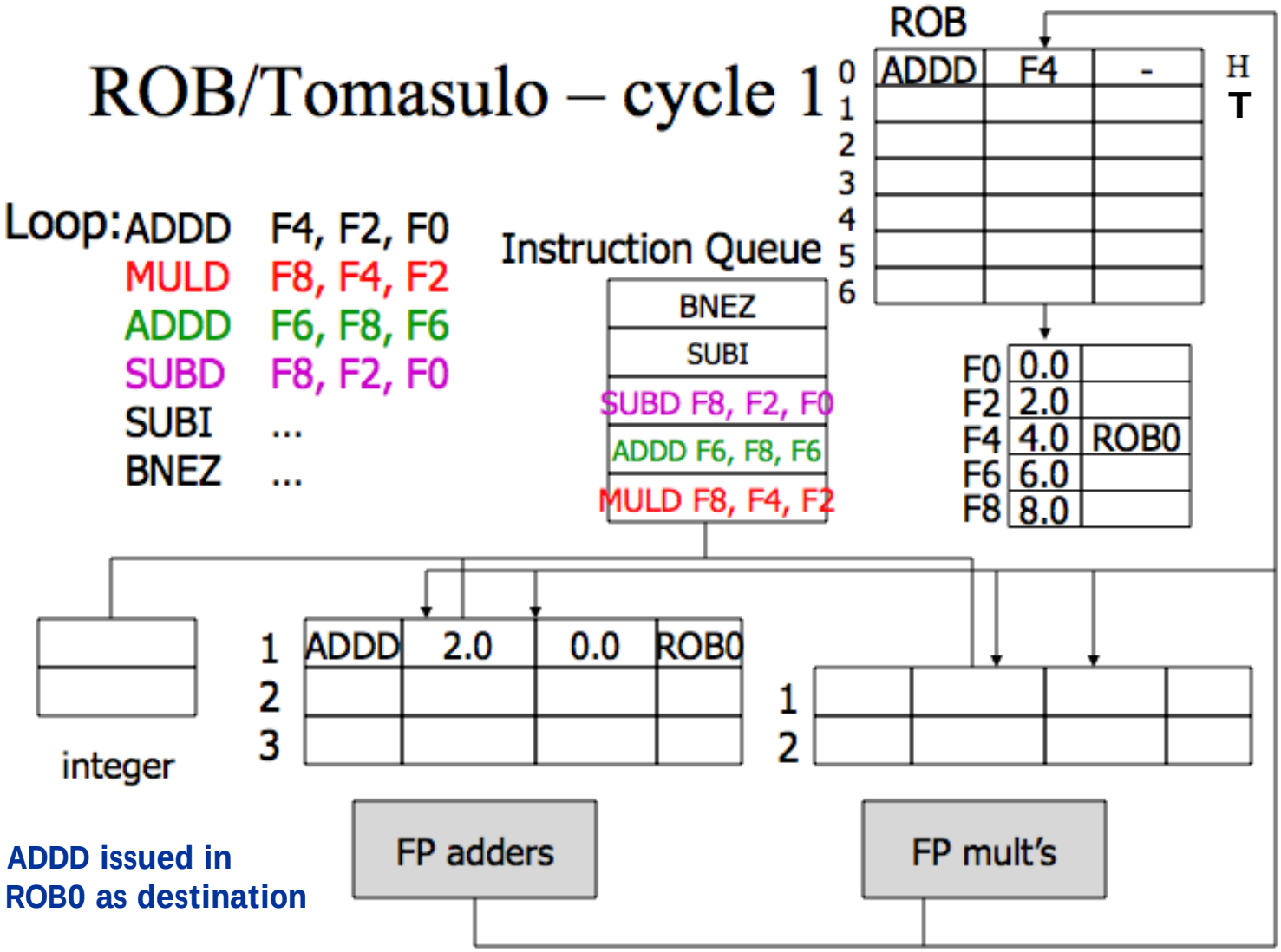
Each entry in **ROB** contains **four fields:**

- ***Instruction type field*** – indicates whether instruction is a branch (no destination result), a store (has memory address destination), or a load/ALU (register destination)
- ***Destination field***: supplies register number (for loads and ALU instructions) or memory address (for stores) where results should be written;
- ***Value field***: used to hold value of result until instruction commits
- ***Ready field***: indicates that instruction has completed execution, value is ready

Tomasulo with ReOrder Buffer

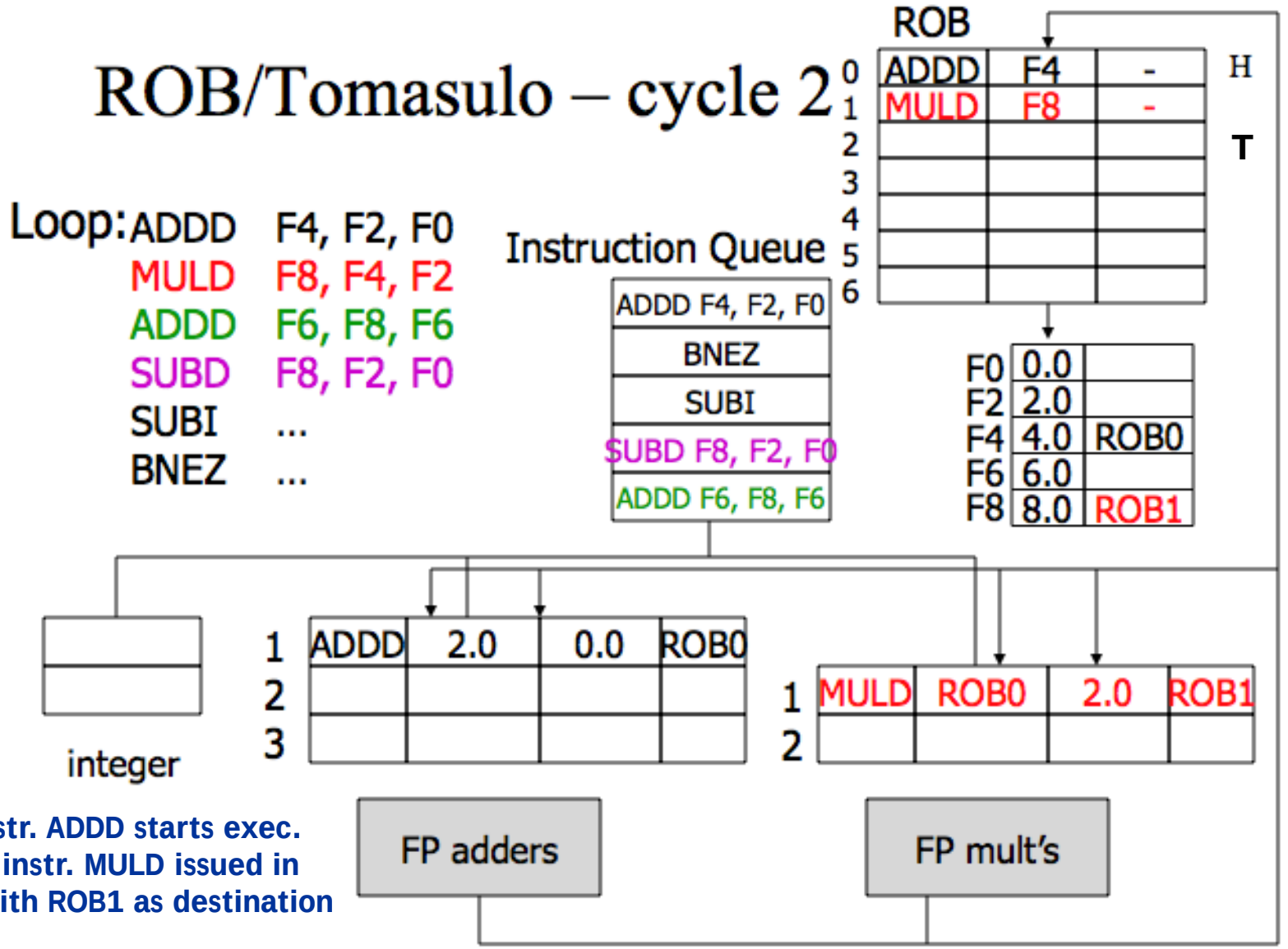


Tomasulo with ReOrder Buffer



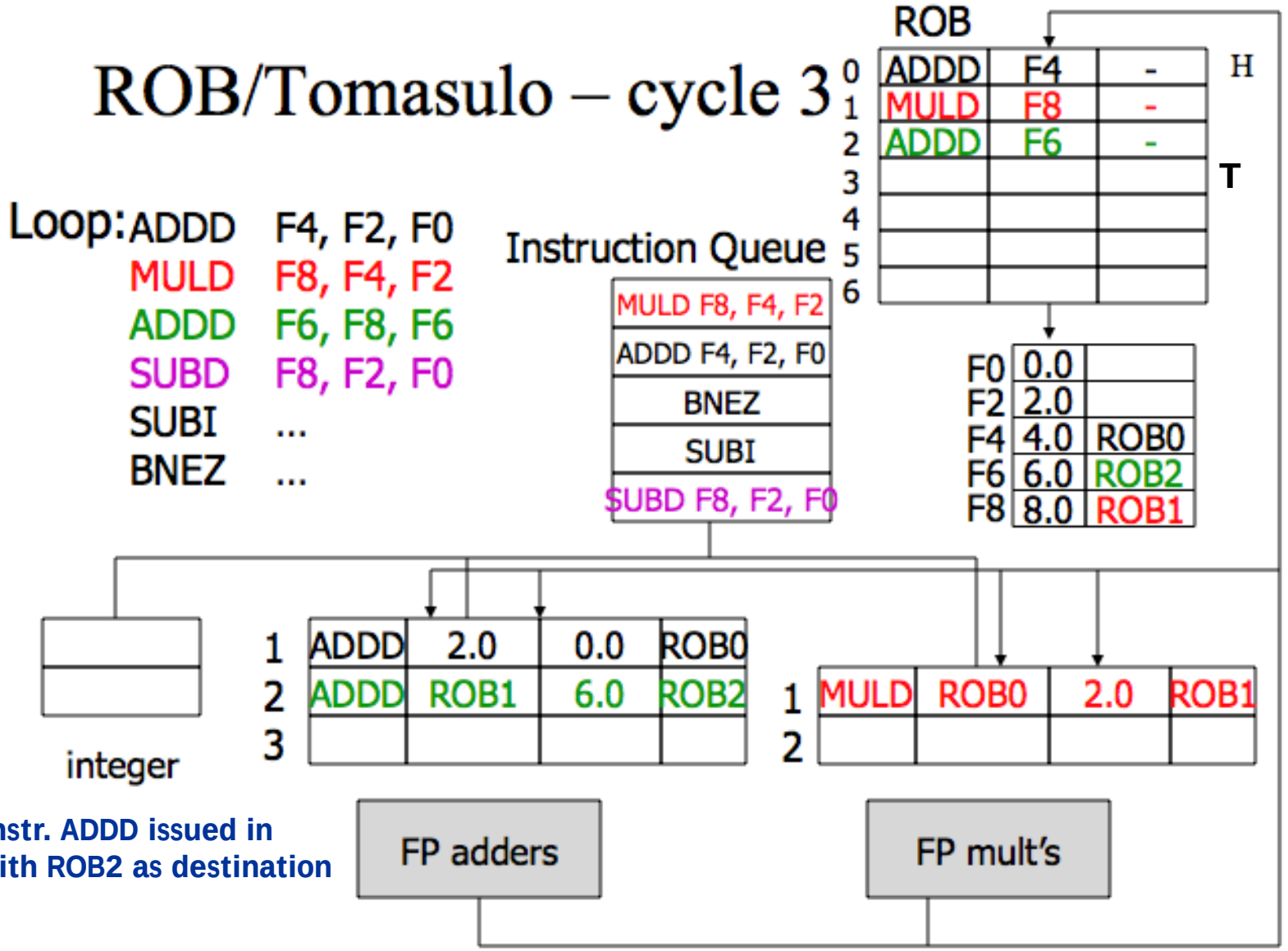
First instr. ADDD issued in RSA1 with ROB0 as destination

Tomasulo with ReOrder Buffer



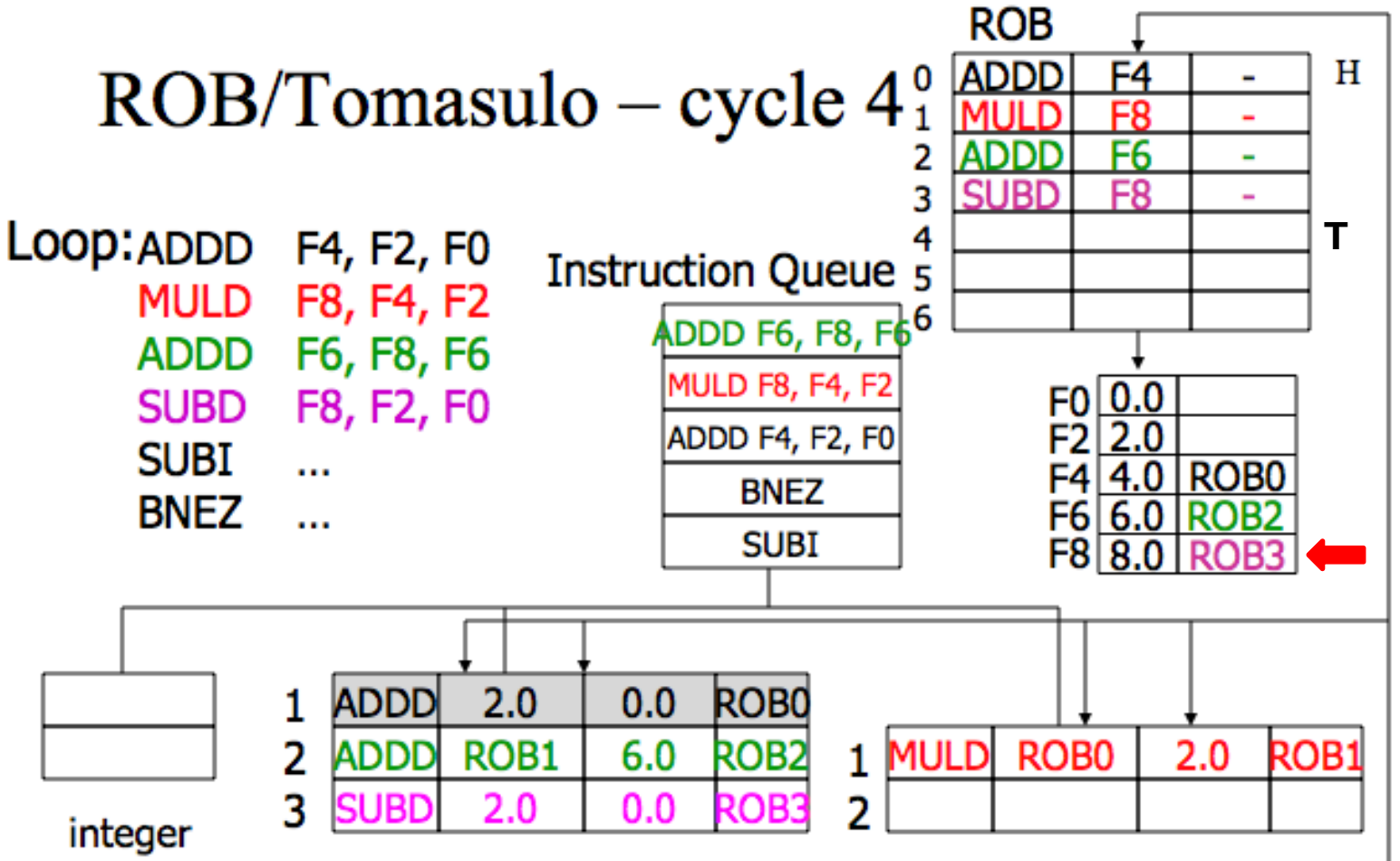
First instr. ADDD starts exec.
Second instr. MULD issued in RSM1 with ROB1 as destination

Tomasulo with ReOrder Buffer



Third instr. ADDD issued in RSA2 with ROB2 as destination

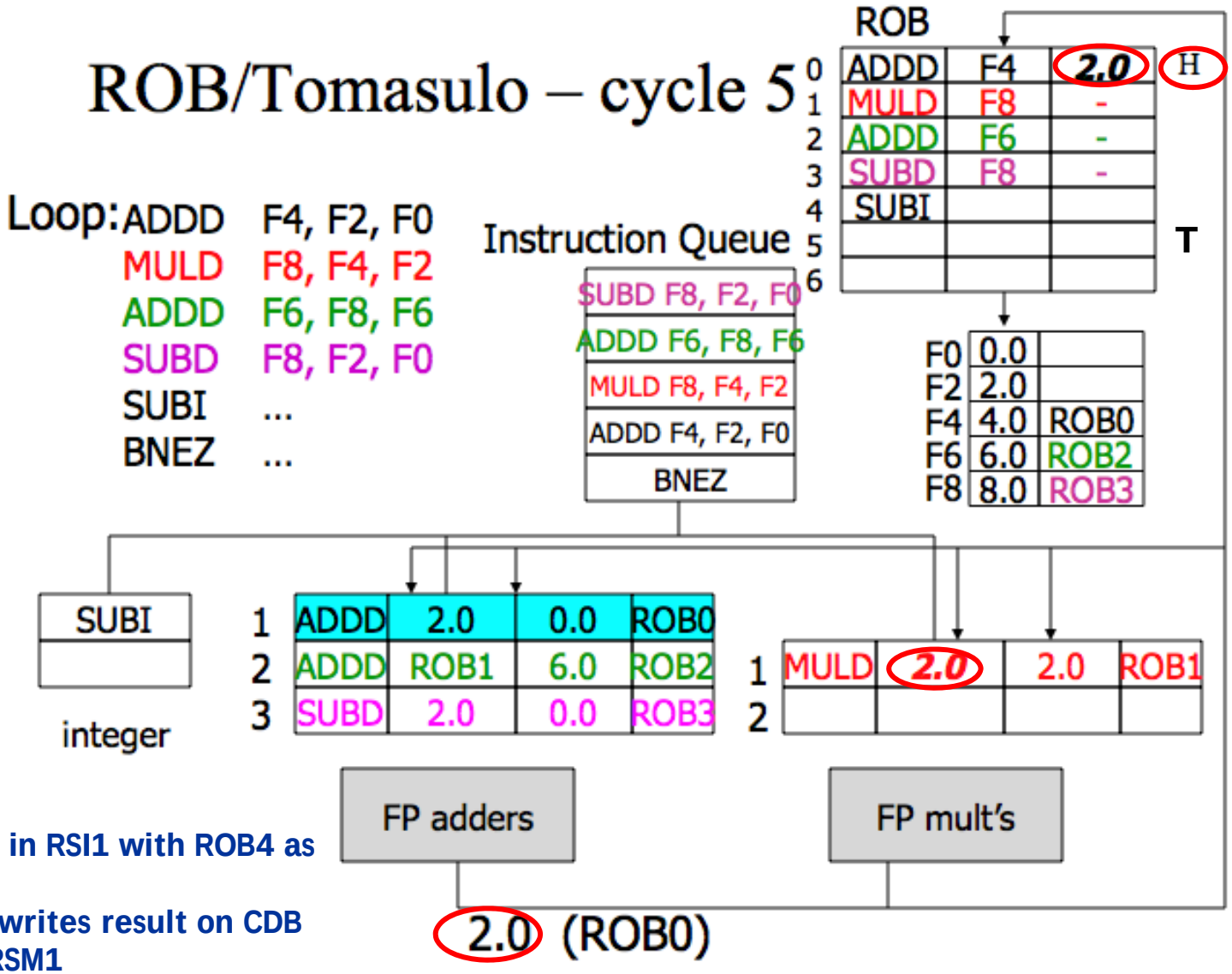
Tomasulo with ReOrder Buffer



Fourth instr. SUBD issued in RSA3 with ROB3 as destination (WAW on F8 solved)

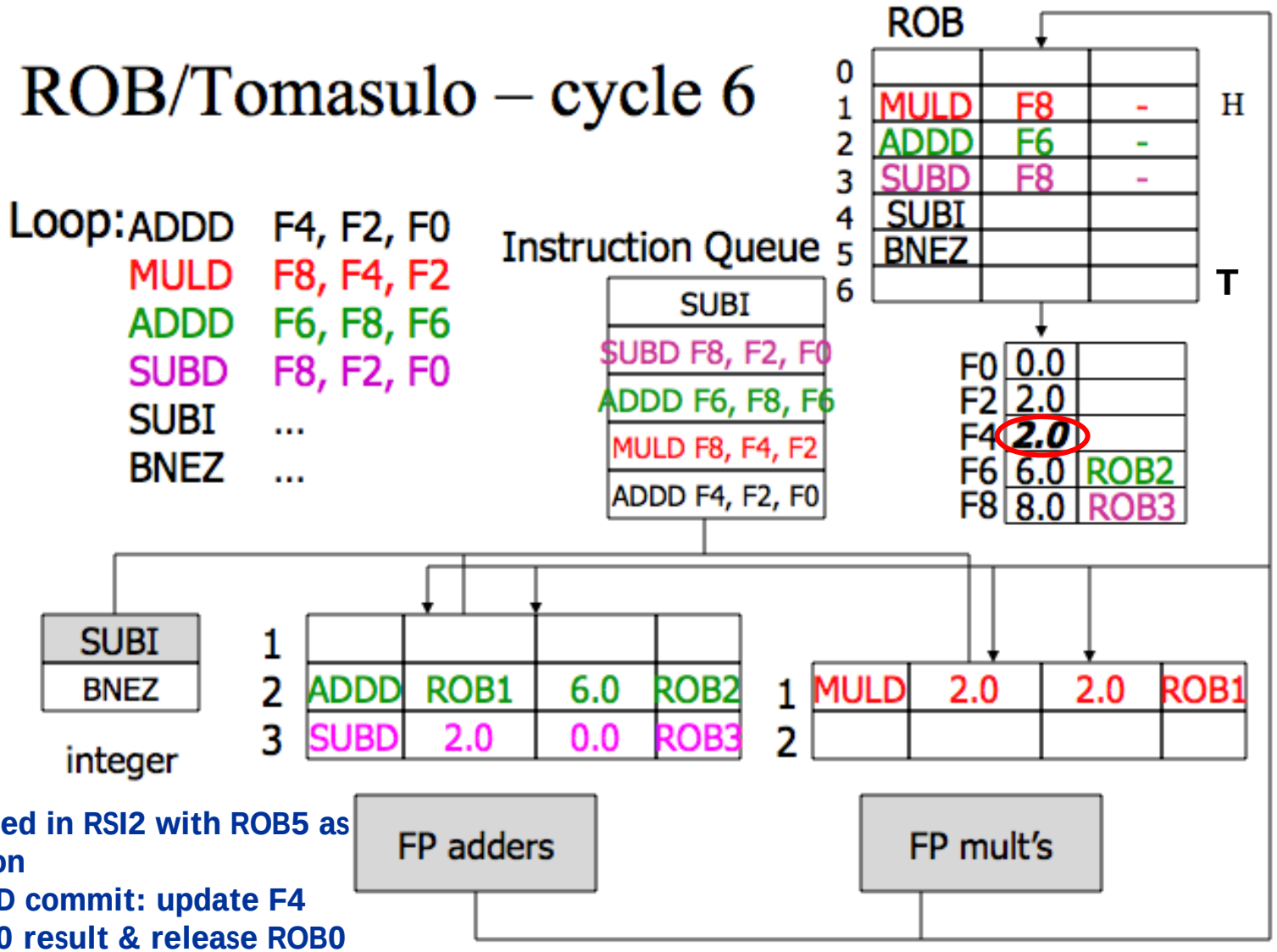
First ADDD exec. completed

Tomasulo with ReOrder Buffer



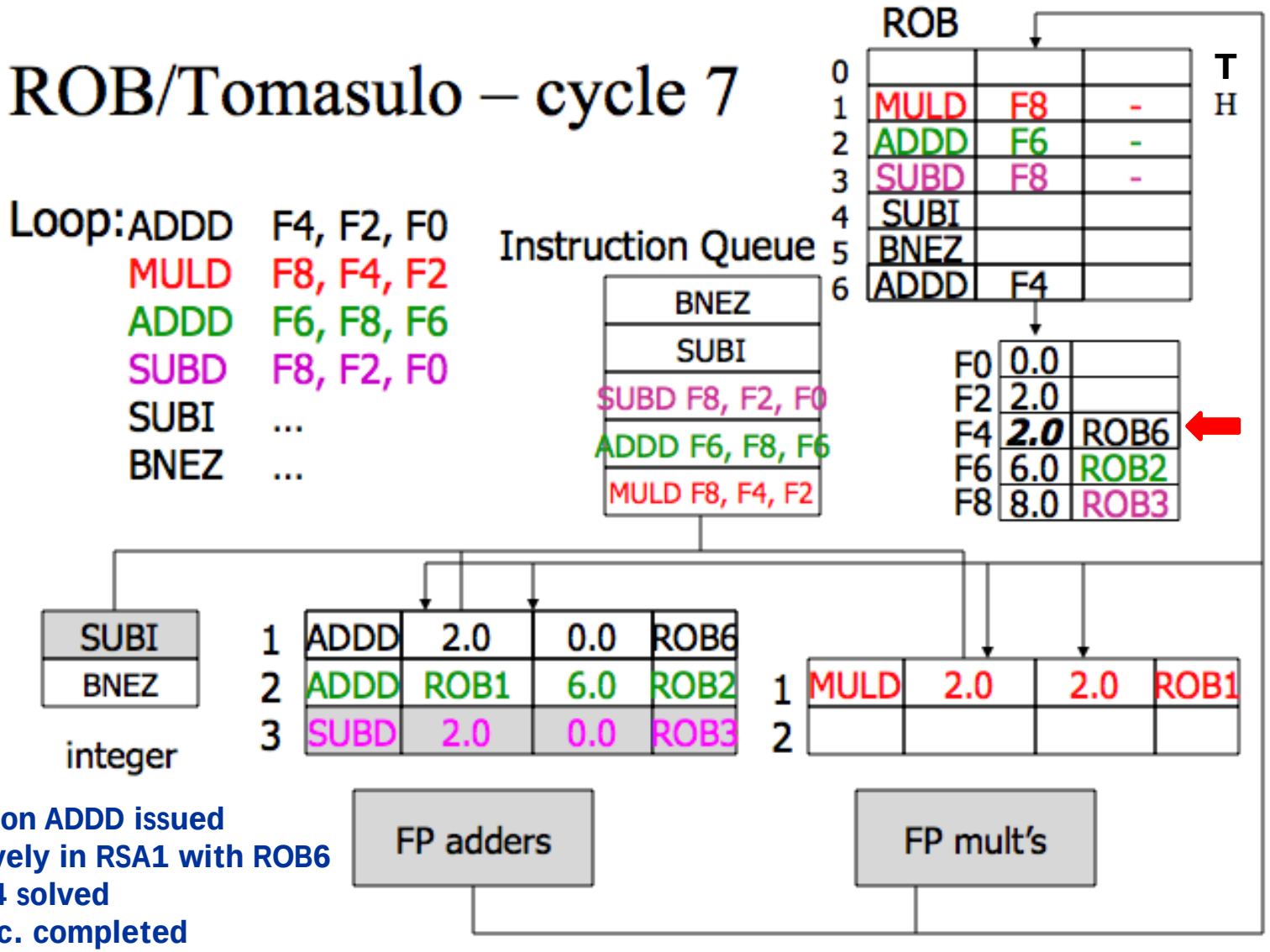
SUBI issued in RSI1 with ROB4 as destination
First ADDD writes result on CDB & ROB0 & RSM1

Tomasulo with ReOrder Buffer



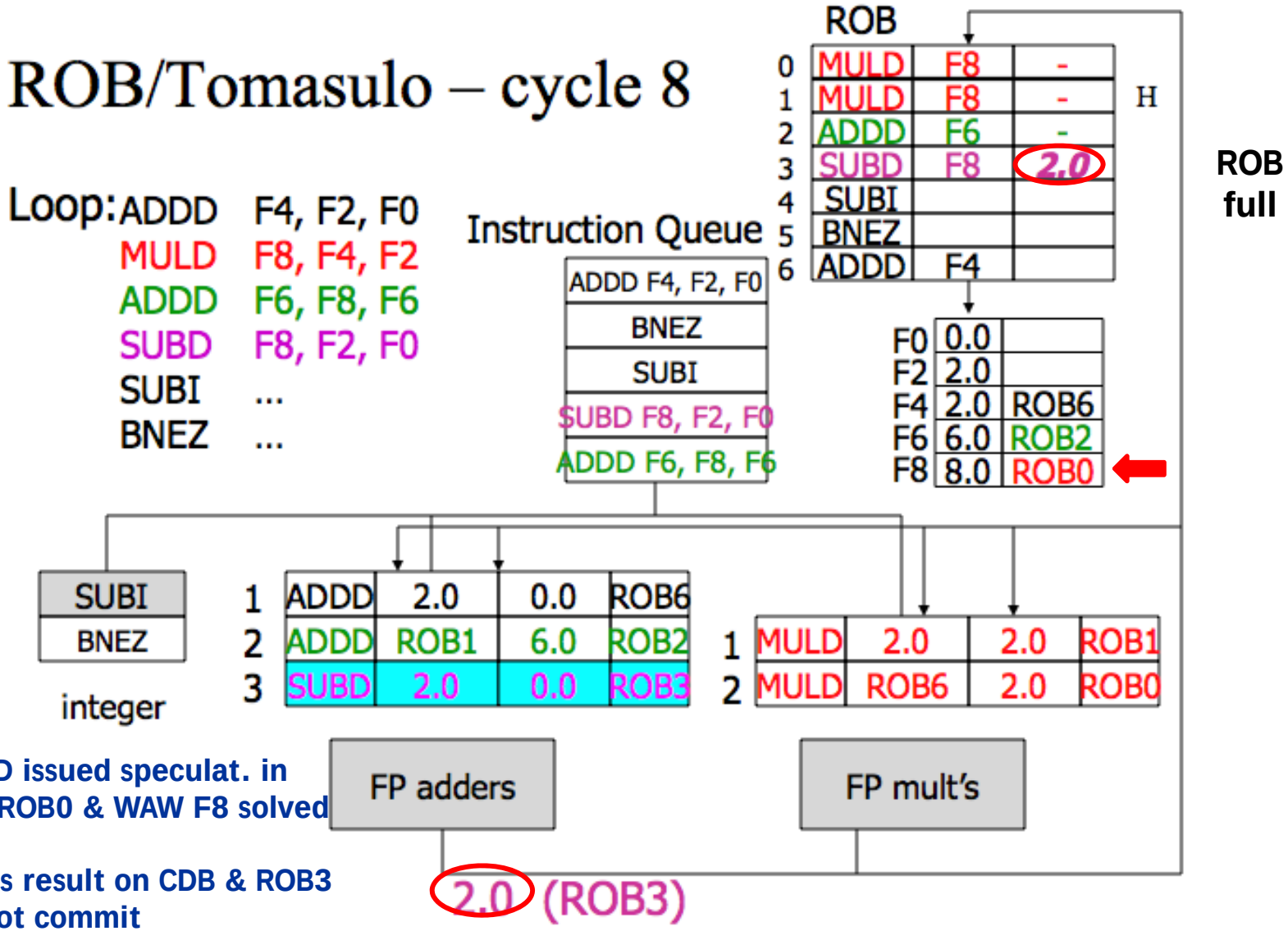
BNEZ issued in RSI2 with ROB5 as destination
First ADDD commit: update F4 with ROB0 result & release ROB0

Tomasulo with ReOrder Buffer



2nd iteration ADDD issued speculatively in RSA1 with ROB6 & WAW F4 solved
SUBD exec. completed

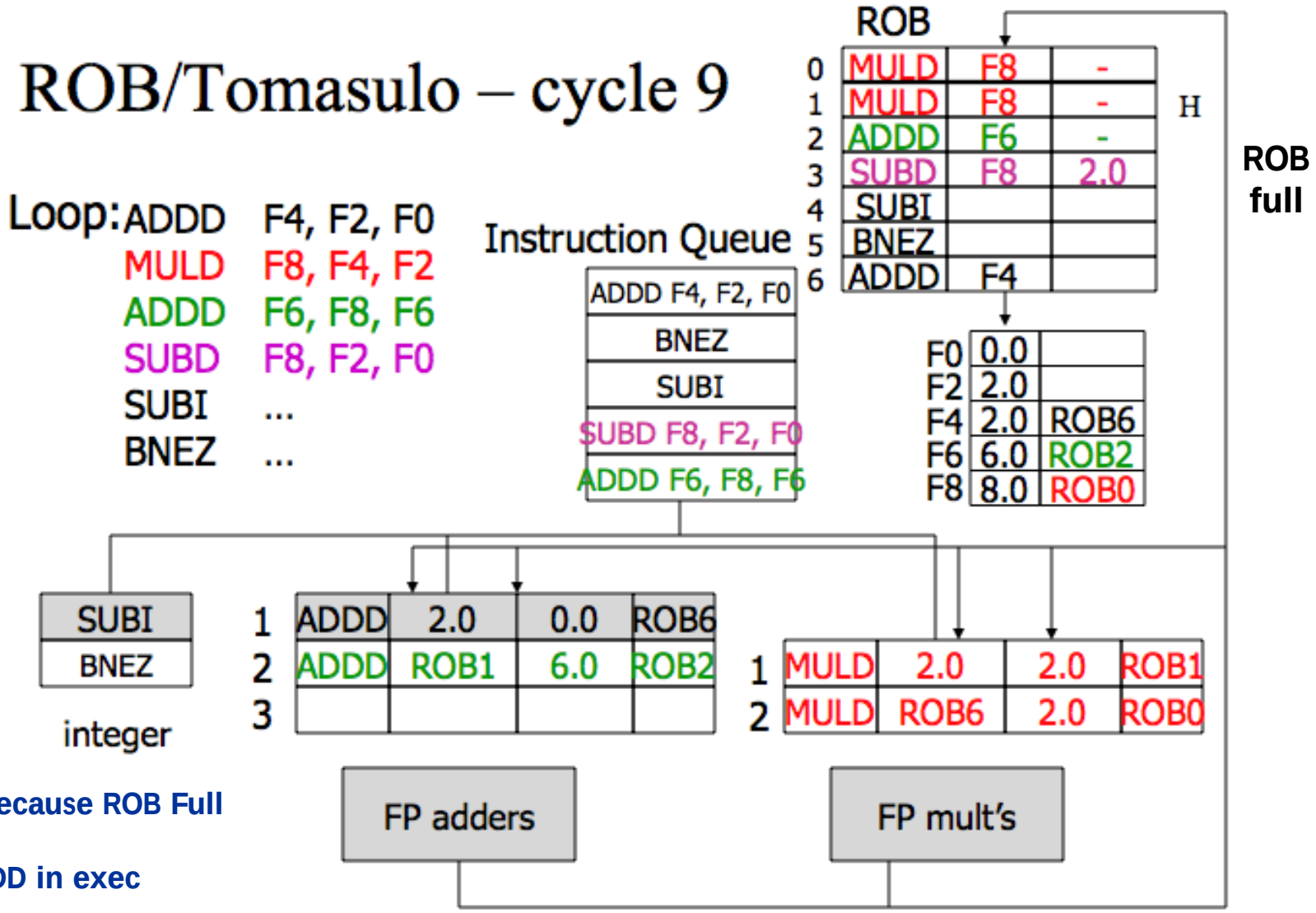
Tomasulo with ReOrder Buffer



2nd it. MULD issued speculat. in RSM2 with ROB0 & WAW F8 solved

SUBD writes result on CDB & ROB3 but it cannot commit

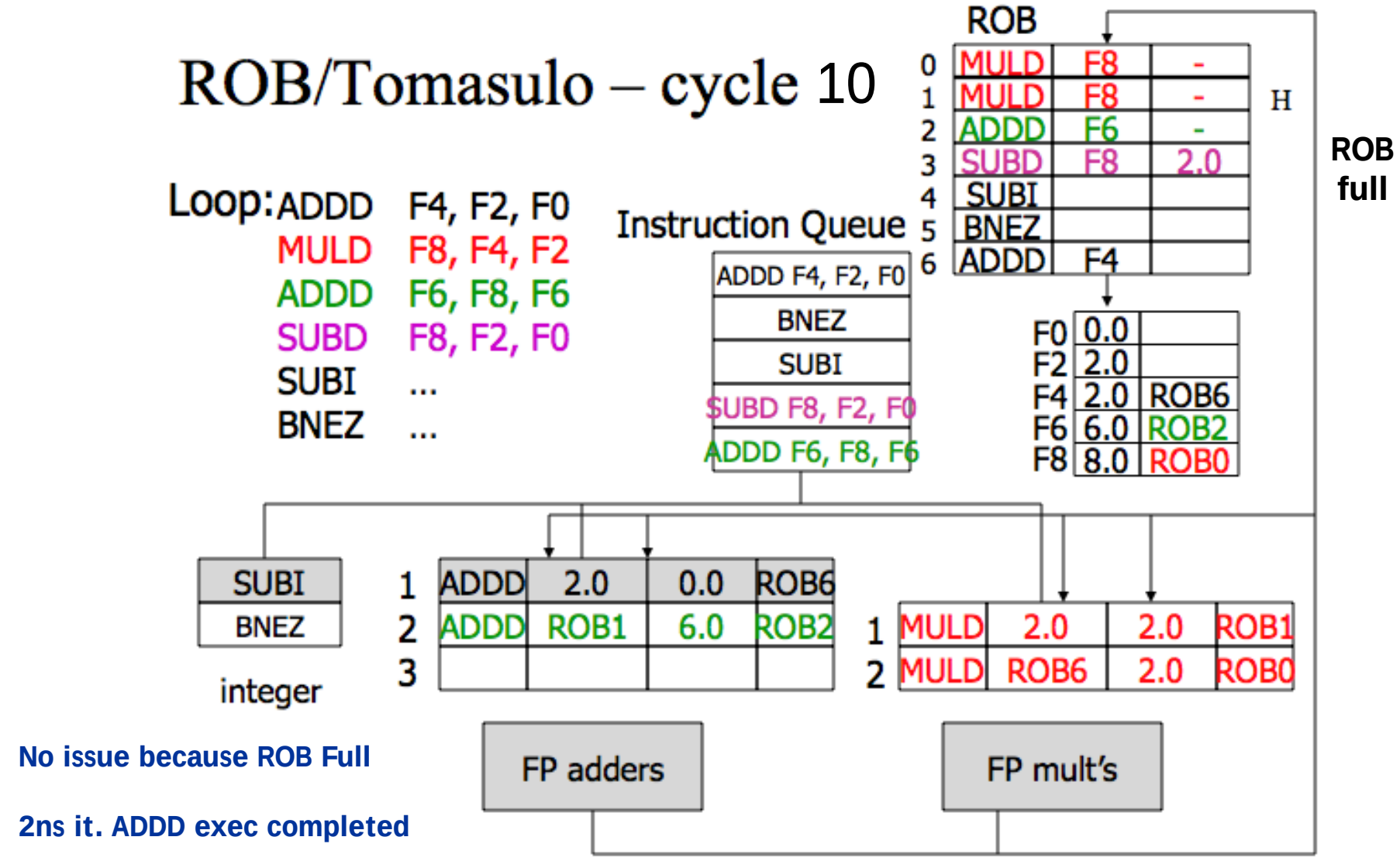
Tomasulo with ReOrder Buffer



No issue because ROB Full

2ns it. ADDD in exec

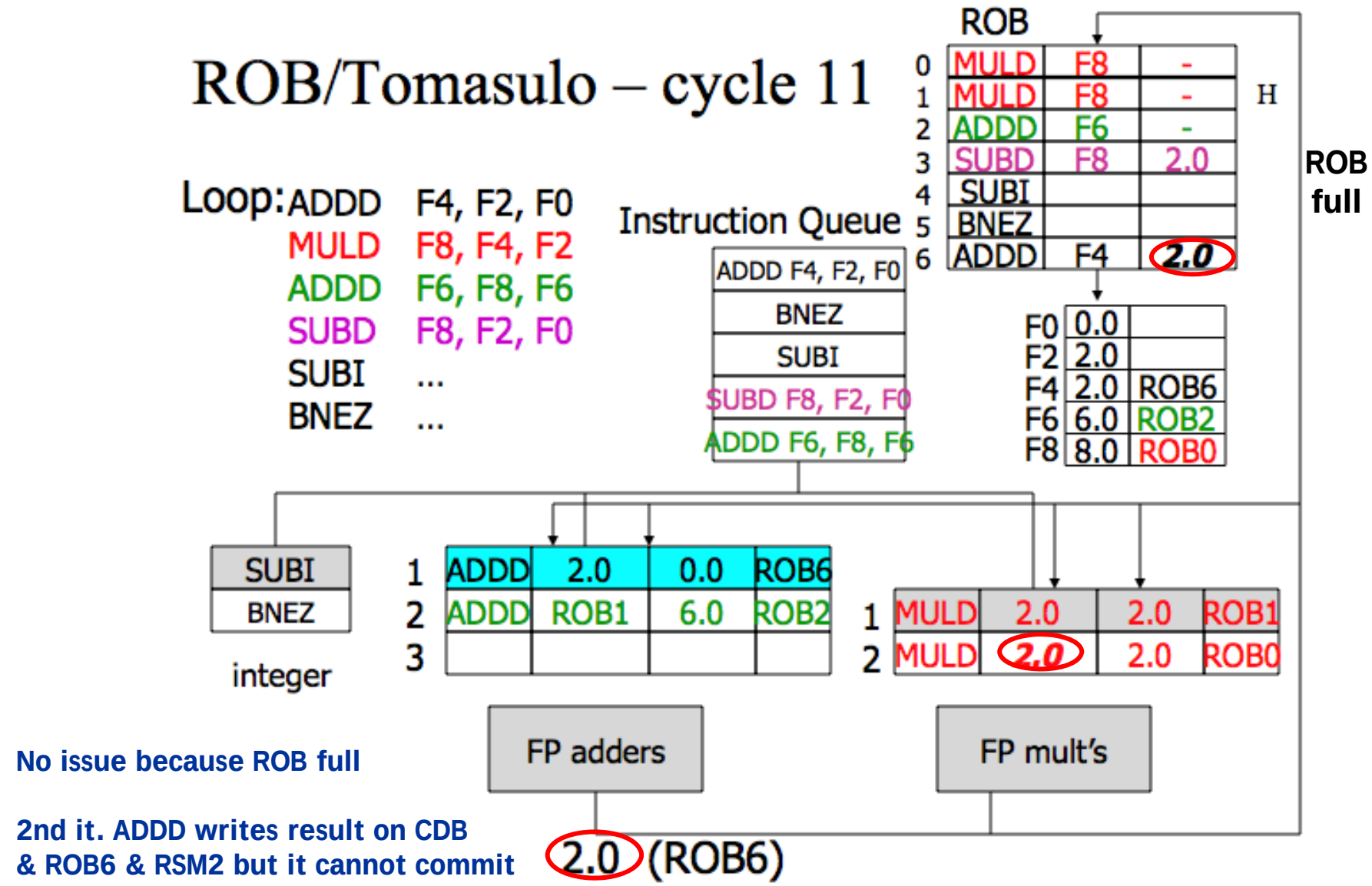
Tomasulo with ReOrder Buffer



No issue because ROB Full

2ns it. ADDD exec completed

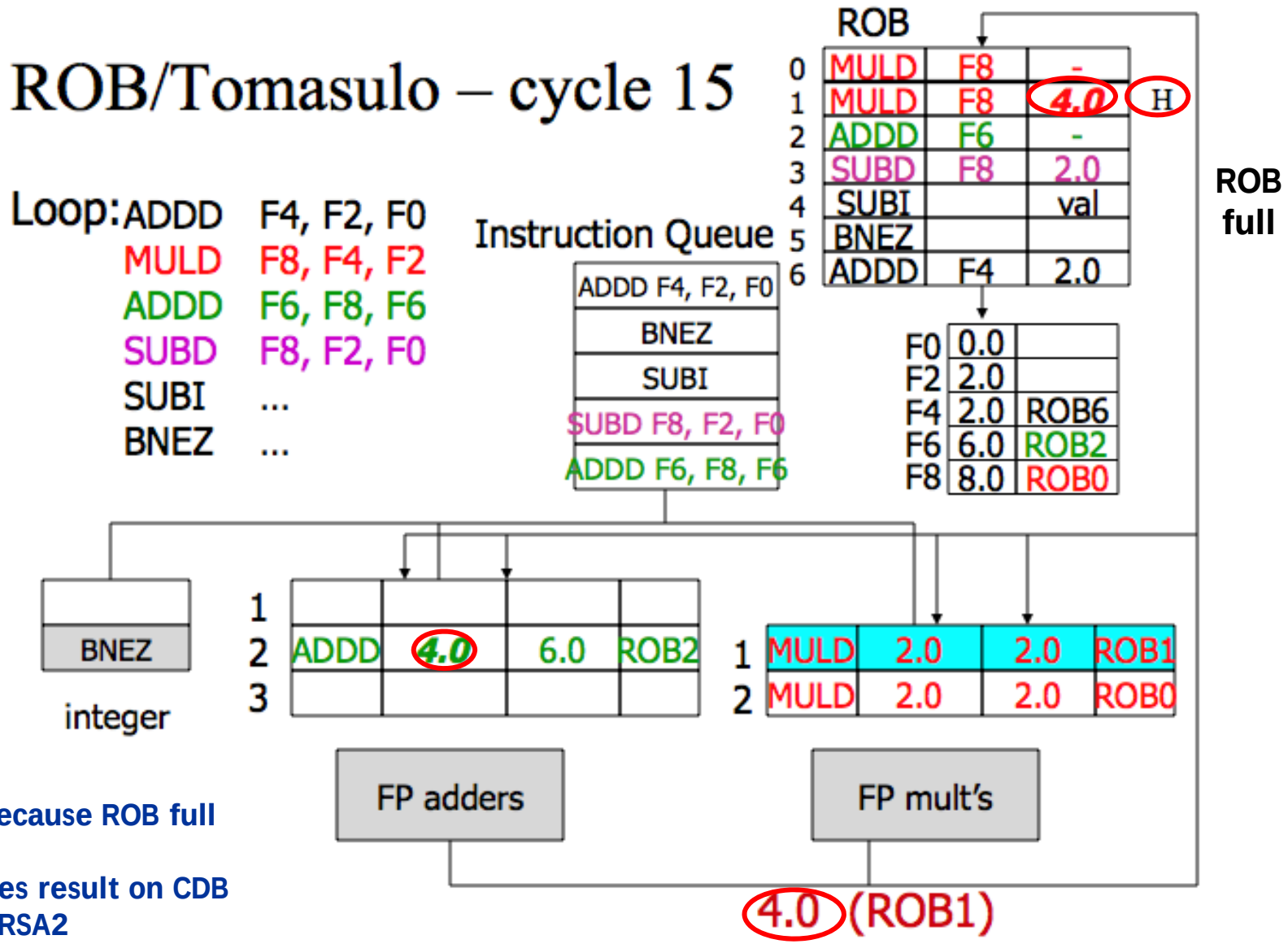
Tomasulo with ReOrder Buffer



No issue because ROB full

2nd it. ADDD writes result on CDB & ROB6 & RSM2 but it cannot commit

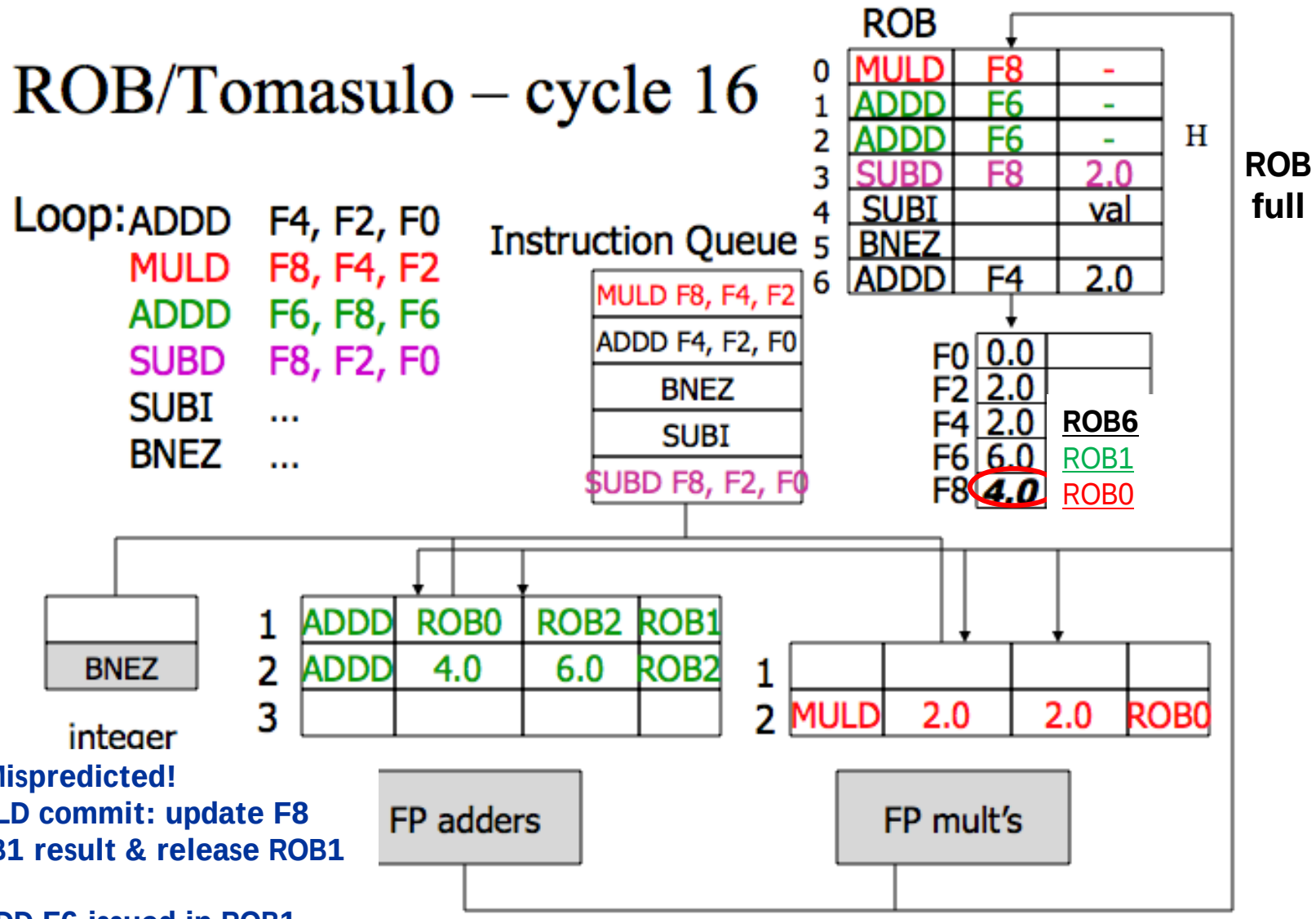
Tomasulo with ReOrder Buffer



No issue because ROB full

MULD writes result on CDB & ROB1 & RSA2

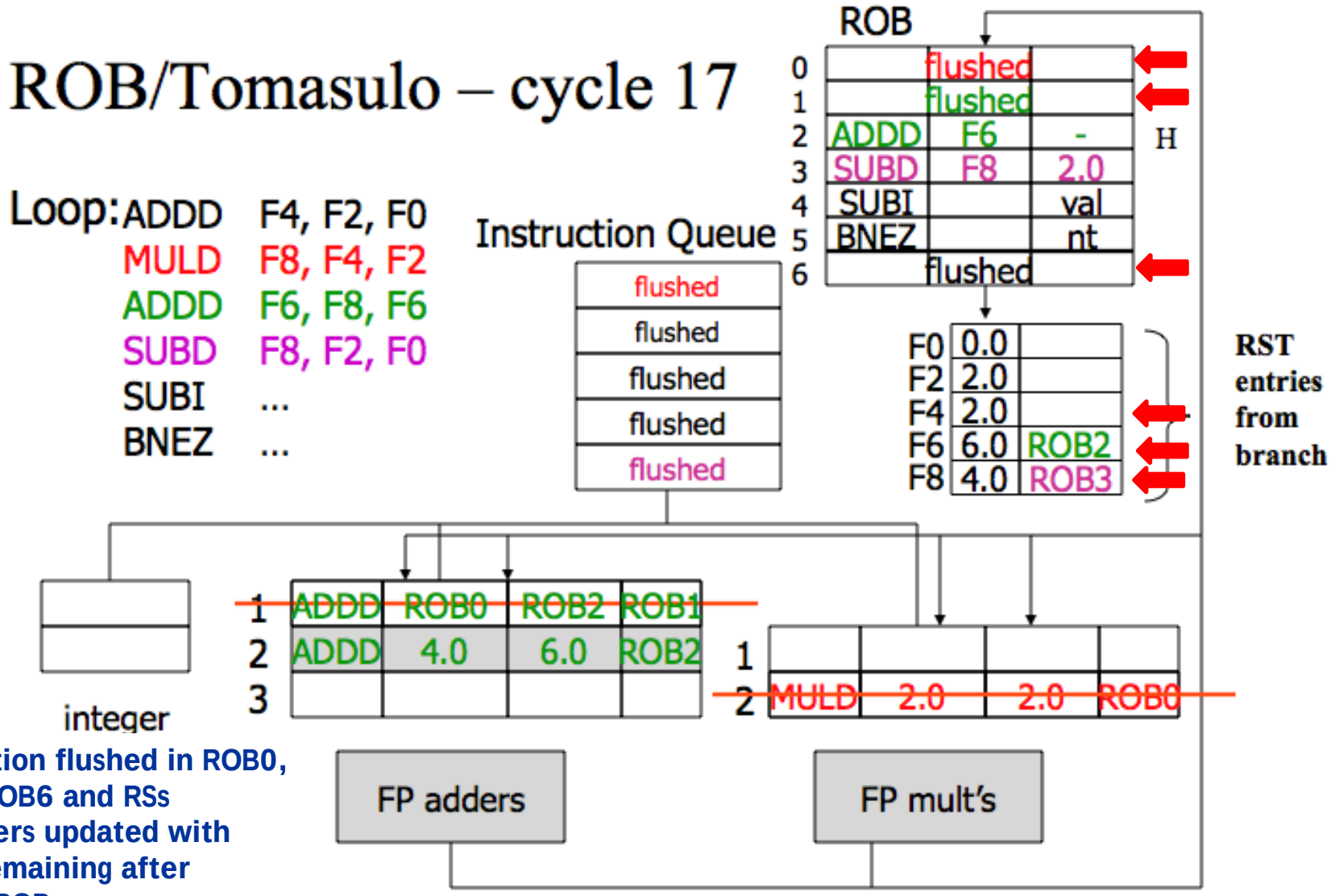
Tomasulo with ReOrder Buffer



Branch Mispredicted!
First MULD commit: update F8 with ROB1 result & release ROB1

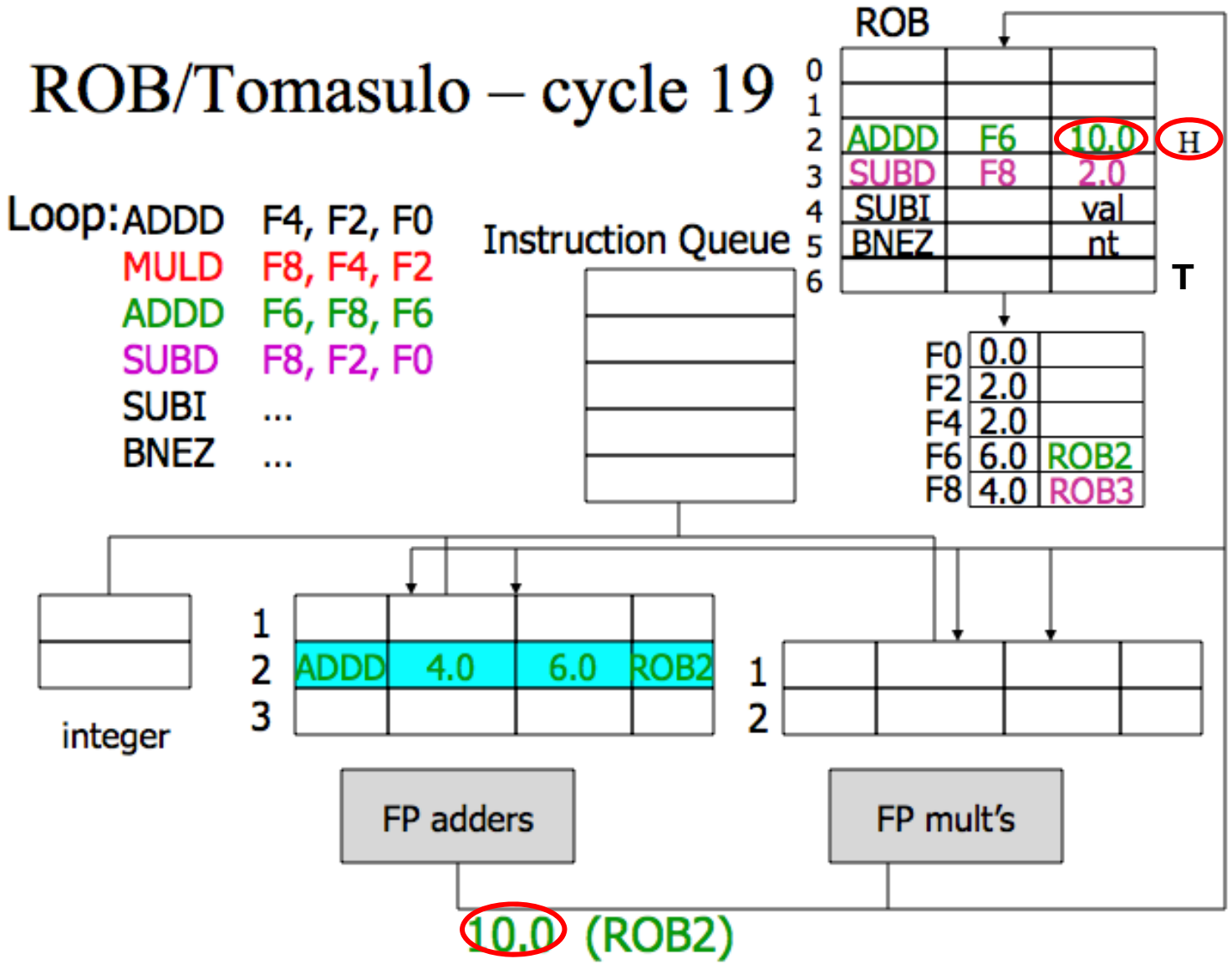
2nd it ADDD F6 issued in ROB1

Tomasulo with ReOrder Buffer

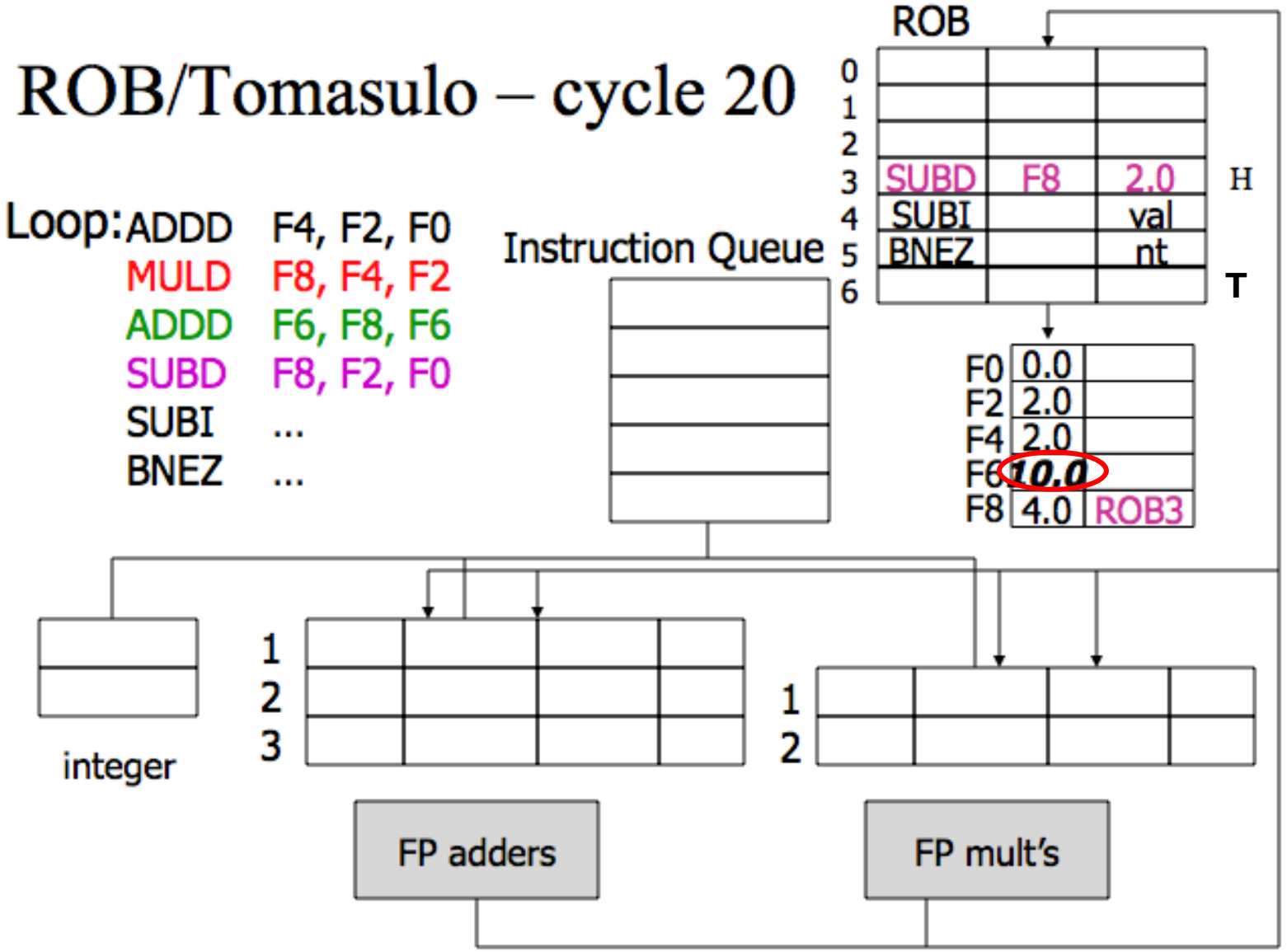


2nd iteration flushed in ROB0, ROB1, ROB6 and RSs
RF pointers updated with values remaining after flushing ROB

Tomasulo with ReOrder Buffer



Tomasulo with ReOrder Buffer



Tomasulo with ReOrder Buffer