

Προηγμένη Αρχιτεκτονική Υπολογιστών

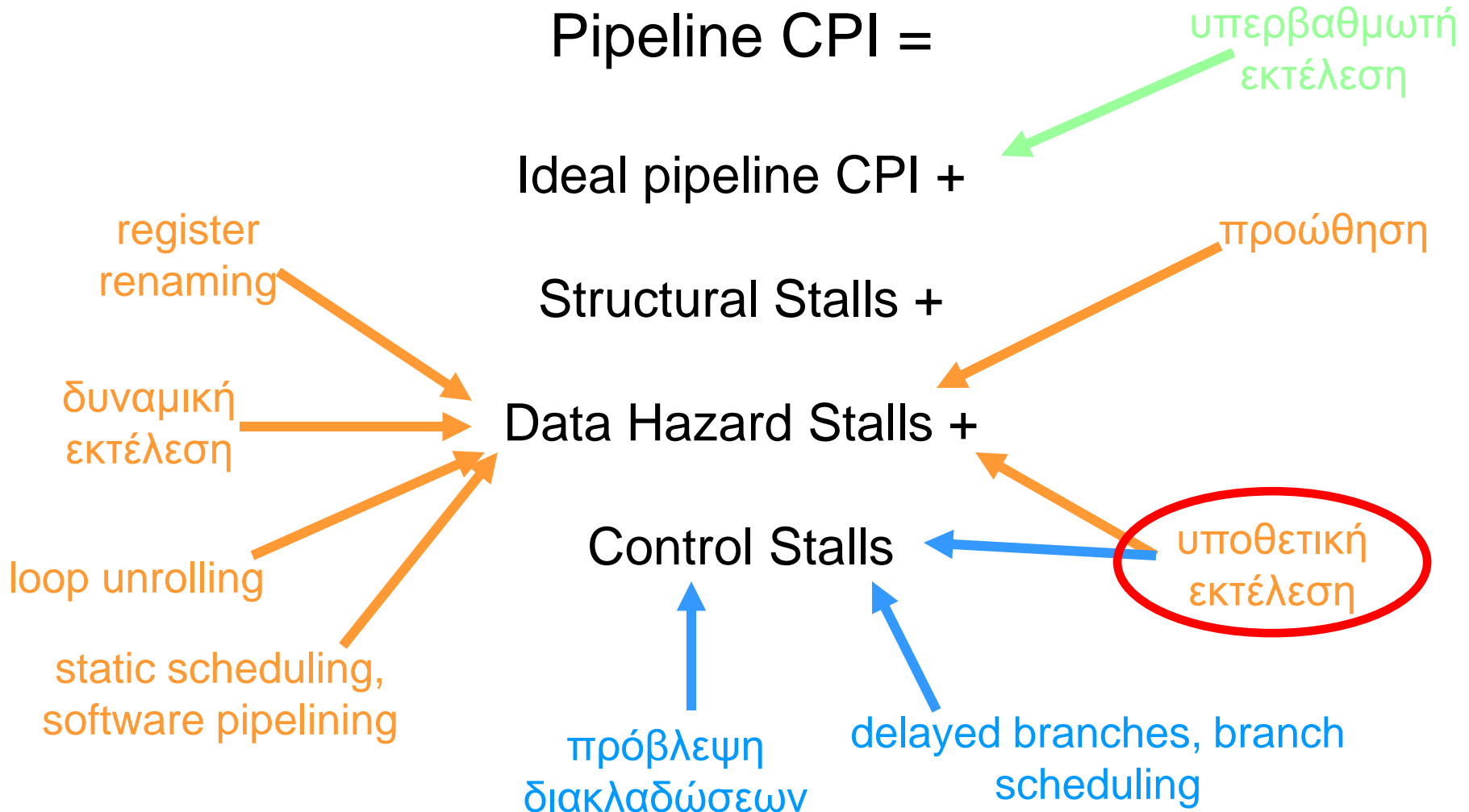
Υποθετική Εκτέλεση Εντολών (Hardware-Based Speculation)

Νεκτάριος Κοζύρης & Διονύσης Πνευματικάτος
{nkoziris,pnevmati}@cslab.ece.ntua.gr

8ο εξάμηνο ΣΗΜΜΥ – Ακαδημαϊκό Έτος: 2019-20

<http://www.cslab.ece.ntua.gr/courses/advcomparch/>

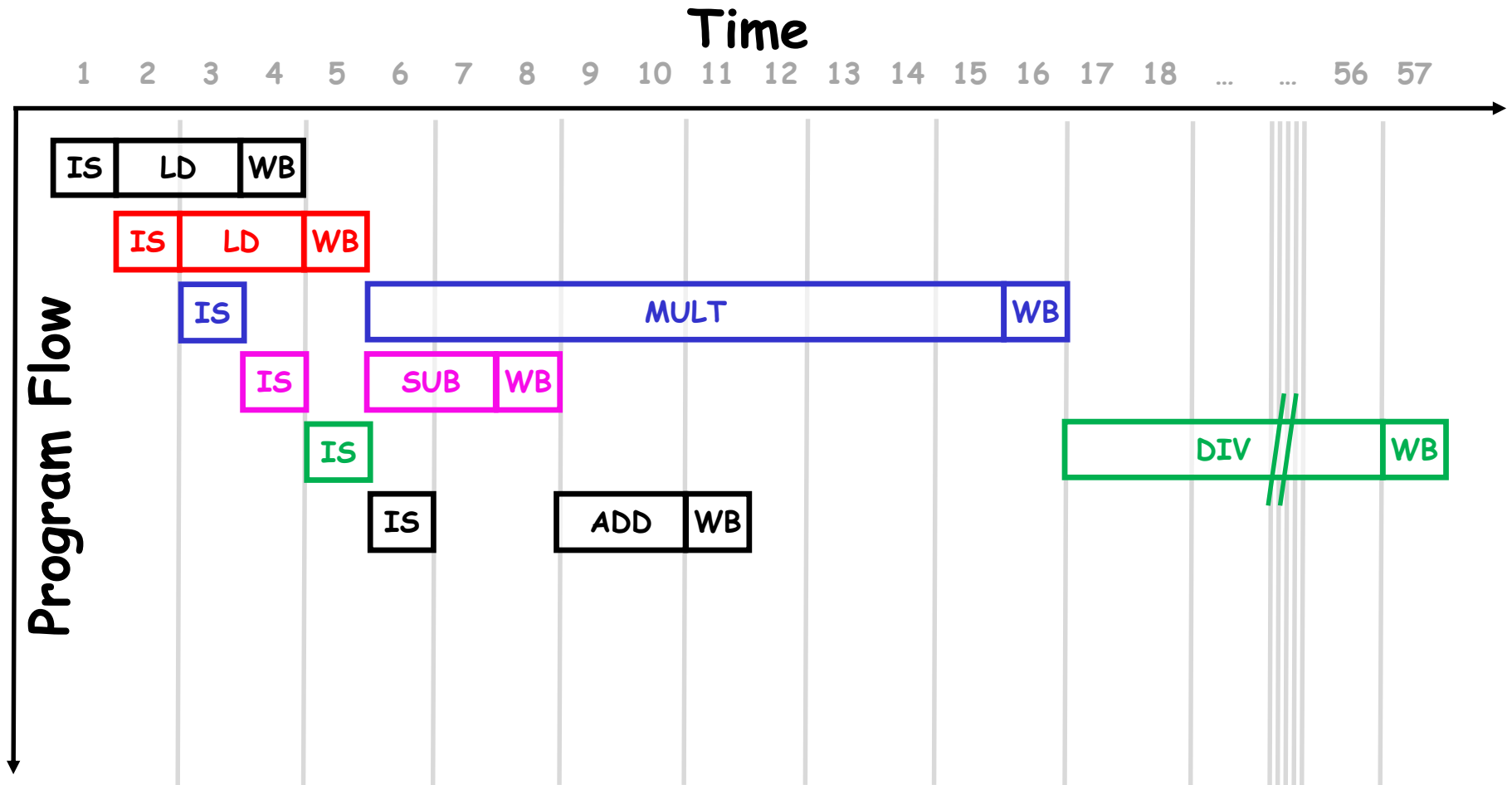
Τεχνικές βελτίωσης του CPI



Δυναμική Δρομολόγηση Εντολών

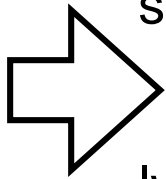
- Tomasulo, Explicit Register Renaming
 - in-order issue
 - out-of-order execution
 - **out-of-order completion**
- Περισσότερος παραλληλισμός (ILP)
- Βελτίωση της απόδοσης του συστήματος
- **Προβλήματα**
 - Interrupts/Exceptions
 - Εντολές διακλάδωσης (branches)

Tomasulo Out of Order Completion



Device Interrupt

Network Interrupt



```
add    r1,r2,r3
subi   r4,r1,#4
slli   r4,r4,#2
      (!)
lw     r2,0(r4)
lw     r3,4(r4)
add    r2,r2,r3
sw     8(r4),r2
```

Σώσε PC
Απενεργ. Ints
Supervisor Mode

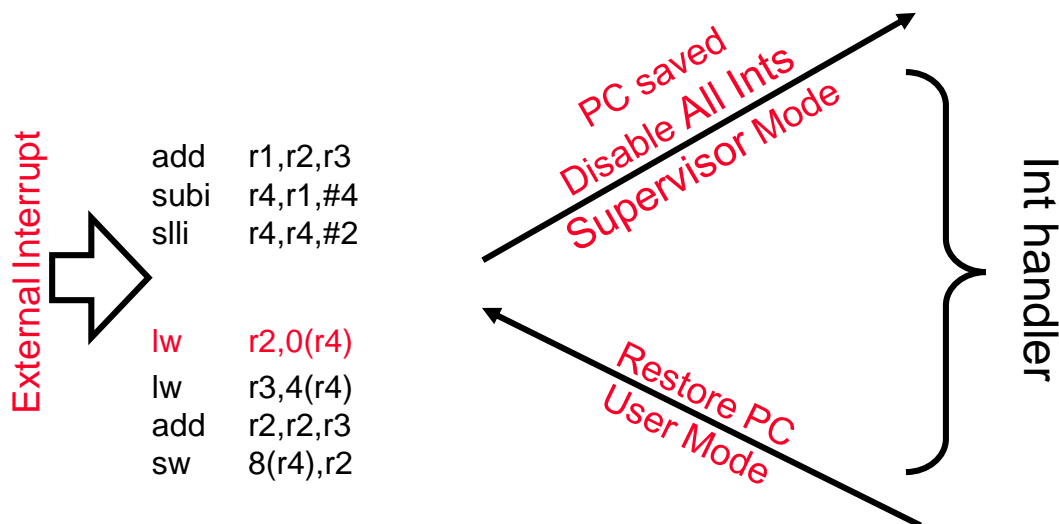
Interrupt Handler

Μεγάλωσε priority
Ενεργοποίηση Ints
Σώσε registers
lw r1,20(r0)
lw r2,0(r1)
addi r3,r0,#5
sw 0(r1),r3
Επανάφερε registers
Καθάρισε Int
Απενεργ. Ints
Επανάφερε priority
RTE

Επανάφερε PC
User Mode

Precise Interrupts/Exceptions


- Ένα interrupt ή exception ονομάζεται **precise** εάν υπάρχει μία εντολή (ή interrupt point) για το οποίο:
 - Όλες οι προηγούμενες εντολές έχουν πλήρως εκτελεστεί.
 - Καμία εντολή (μαζί με την interrupting instruction) δεν έχει αλλάξει την κατάσταση της μηχανής.
- Αυτό σημαίνει ότι μπορούμε να επανεκκινήσουμε την εκτέλεση από το interrupt point και “να πάρουμε τα σωστά αποτελέσματα”
 - Στο παράδειγμά μας: Interrupt point είναι η **lw** εντολή



Γιατί χρειαζόμαστε precise interrupts?

- Αρκετά interrupts/exceptions χρειάζονται να είναι restartable
 - π.χ. TLB faults: διόρθωση translation και επανάληψη του load/store
 - IEEE underflows, illegal operations
- Η επανεκκίνηση δεν απαιτεί preciseness. Ωστόσο, με preciseness είναι **πολύ πιο εύκολη!**
- Απλοποιεί το λειτουργικό σύστημα
 - το process state που χρειάζεται να αποθηκευτεί είναι μικρότερο
 - η επανεκκίνηση γίνεται γρήγορα (καλό για interrupts μικρής διάρκειας/μεγάλης συχνότητας)

Πού είναι το πρόβλημα?

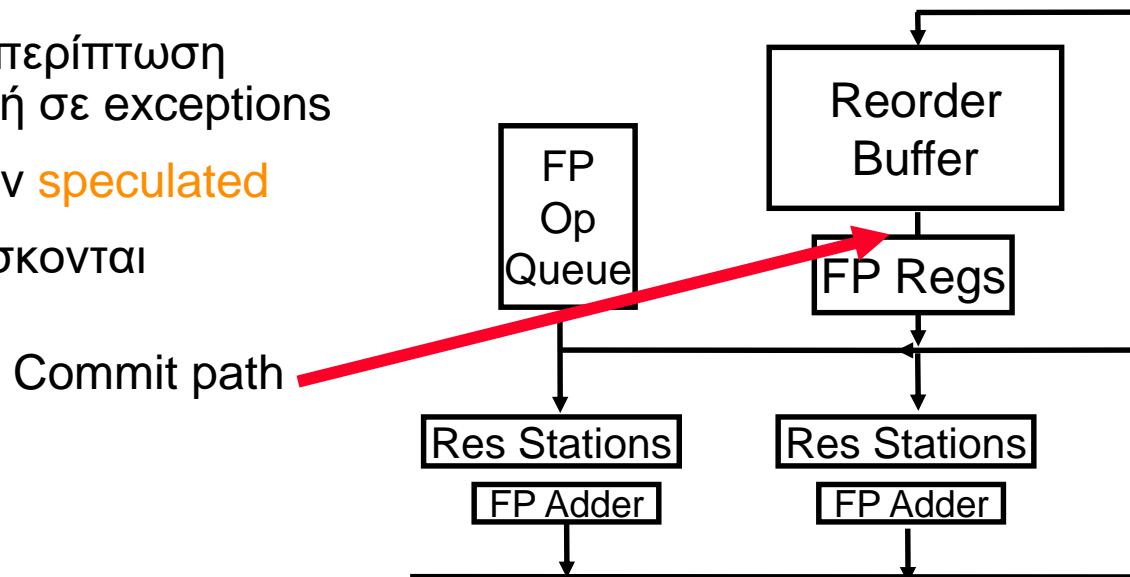
- Όλες οι τεχνικές που είδαμε μέχρι τώρα (Tomasulo, Explicit Register Renaming) υλοποιούν **out-of-order completion**
 - **Imprecise** και όχι **Precise** Interrupts/Exceptions
 - » Είναι πιθανό όταν εμφανισθεί ένα Interrupt/Exception η κατάσταση της μηχανής (processor state) να είναι διαφορετική από ότι αν οι εντολές είχαν εκτελεστεί σειριακά  **Imprecise**
- Πρέπει να αλλάξουμε τη λογική του out-of-order completion, ώστε να μπορούμε να ορίζουμε precise interrupt points μέσα στο instruction stream
- Πρέπει να βρούμε τρόπο ώστε να *συγχρονίσουμε* το completion στάδιο των εντολών με την σειρά στο πρόγραμμα (issue-order)
 - Ο απλούστερος τρόπος: **in-order completion**

Εντολές διακλάδωσης (branches)

- Για να βελτιώσουμε την απόδοση του συστήματος χρησιμοποιούμε **branch prediction**
- Επομένως κατά το οοο execution εκτελούμε εντολές οι οποίες εξαρτώνται από το αποτέλεσμα της εντολής διακλάδωσης
- Αν η πρόβλεψη δεν επαληθευτεί, θα πρέπει να κάνουμε rollback στο σημείο όπου κάναμε την πρόβλεψη, διότι οι εντολές στο λανθασμένο predicted path έχουν ήδη εκτελεστεί
 - Αυτό ακριβώς είναι το πρόβλημα και με τα precise exceptions!
- Λύση: **in-order completion**

Hardware Support

- Ιδέα του **Reorder Buffer** (ROB):
 - Κράτα εντολές σε μία FIFO, ακριβώς με την σειρά που γίνονται issue.
 - » Κάθε ROB entry περιέχει: instruction type (branch/store/register op), destination, value, ready field
 - Όταν η εντολή τελειώσει την εκτέλεση, τοποθέτησε τα αποτελέσματα στον ROB.
 - » Παρέχει operands σε άλλες εντολές στο διάστημα μεταξύ execution complete & commit
 - Η εντολή αλλάζει την κατάσταση της μηχανής στο **commit στάδιο** όχι στο WB → in-order commit → οι τιμές στην κεφαλή του ROB αποθηκεύονται στο register file
 - Εύκολη η αναίρεση σε περίπτωση mispredicted branches ή σε exceptions
 - » Απλά **διαγραφή** των **speculated instructions** που βρίσκονται στο ROB



Αλγόριθμος Tomasulo με ROB

Issue — Πάρε εντολή από FP Op Queue

Αν υπάρχει ελεύθερο reservation station & reorder buffer entry, issue instr & send operands & reorder buffer no. for destination

Execution — Εκτέλεσε εντολή στο Ex Unit (EX)

Όταν και οι τιμές και των 2 source regs είναι έτοιμες εκτέλεσε την εντολή; Αν όχι, παρακολούθησε το CDB για το αποτέλεσμα; Όταν και οι 2 τιμές βρίσκονται στο RS, εκτέλεσε την εντολή; (checks RAW)

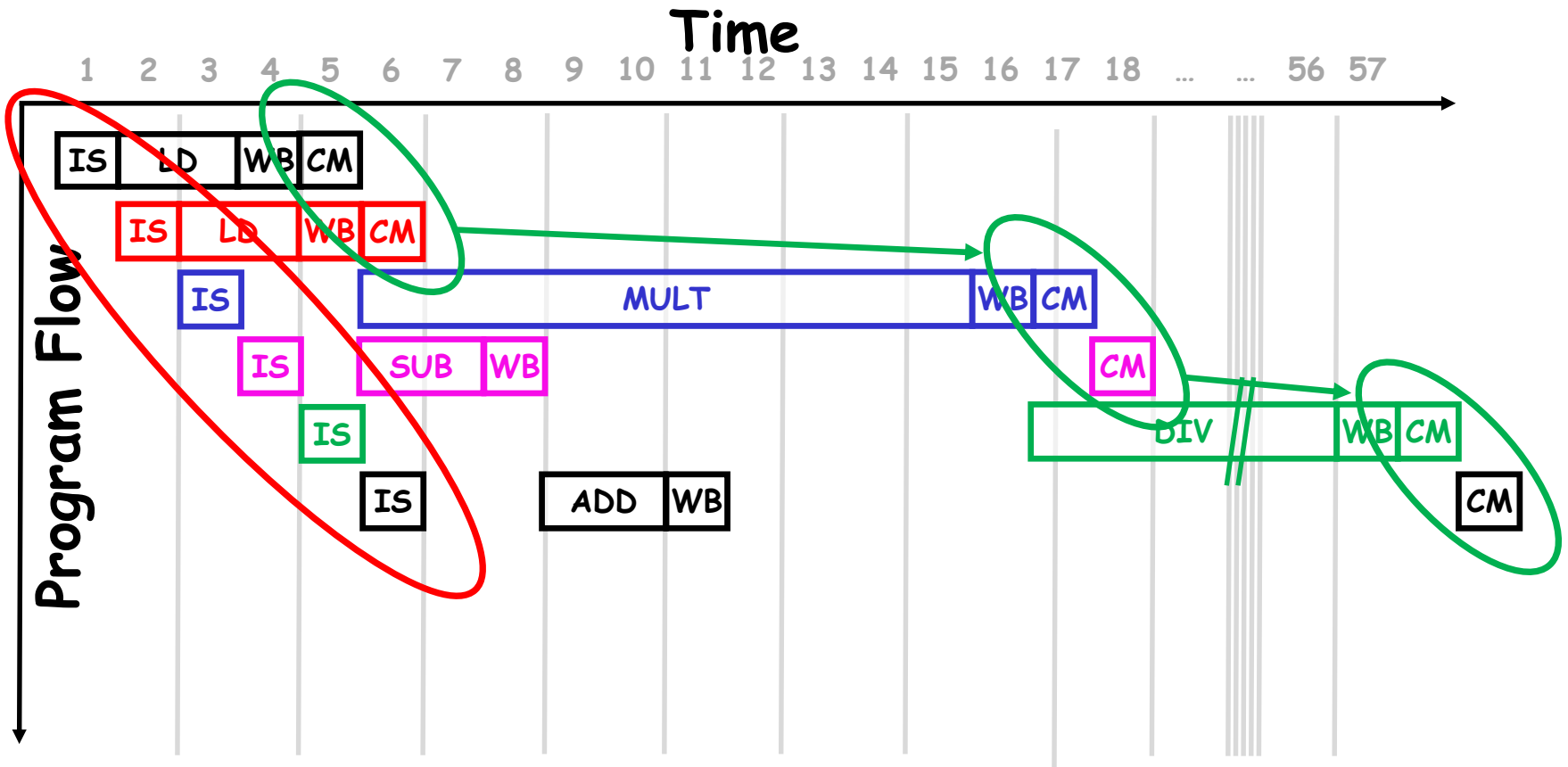
Write result — Τέλος εκτέλεσης (WB)

Γράψε το αποτέλεσμα στο CDB προς όλες τις μονάδες που το περιμένουν & στον reorder buffer; Σημείωσε τον RS ως διαθέσιμο.

Commit — Ανανέωσε τον dest register με το αποτέλεσμα από τον reorder buffer

Όταν η εντολή φτάσει στην κεφαλή του reorder buffer & το result είναι διαθέσιμο, ανανέωσε τον dest register με αυτό (ή, αντίστοιχα, αποθήκευσε στη μνήμη) και βγάλε την εντολή από τον reorder buffer. Όταν η εντολή είναι mispredicted branch, καθάρισε (flush) τον reorder buffer και επανεκκίνησε την εκτέλεση στο σωστό path.

Tomasulo Dynamic Execution



In-order issue

Out-of-order execution & result

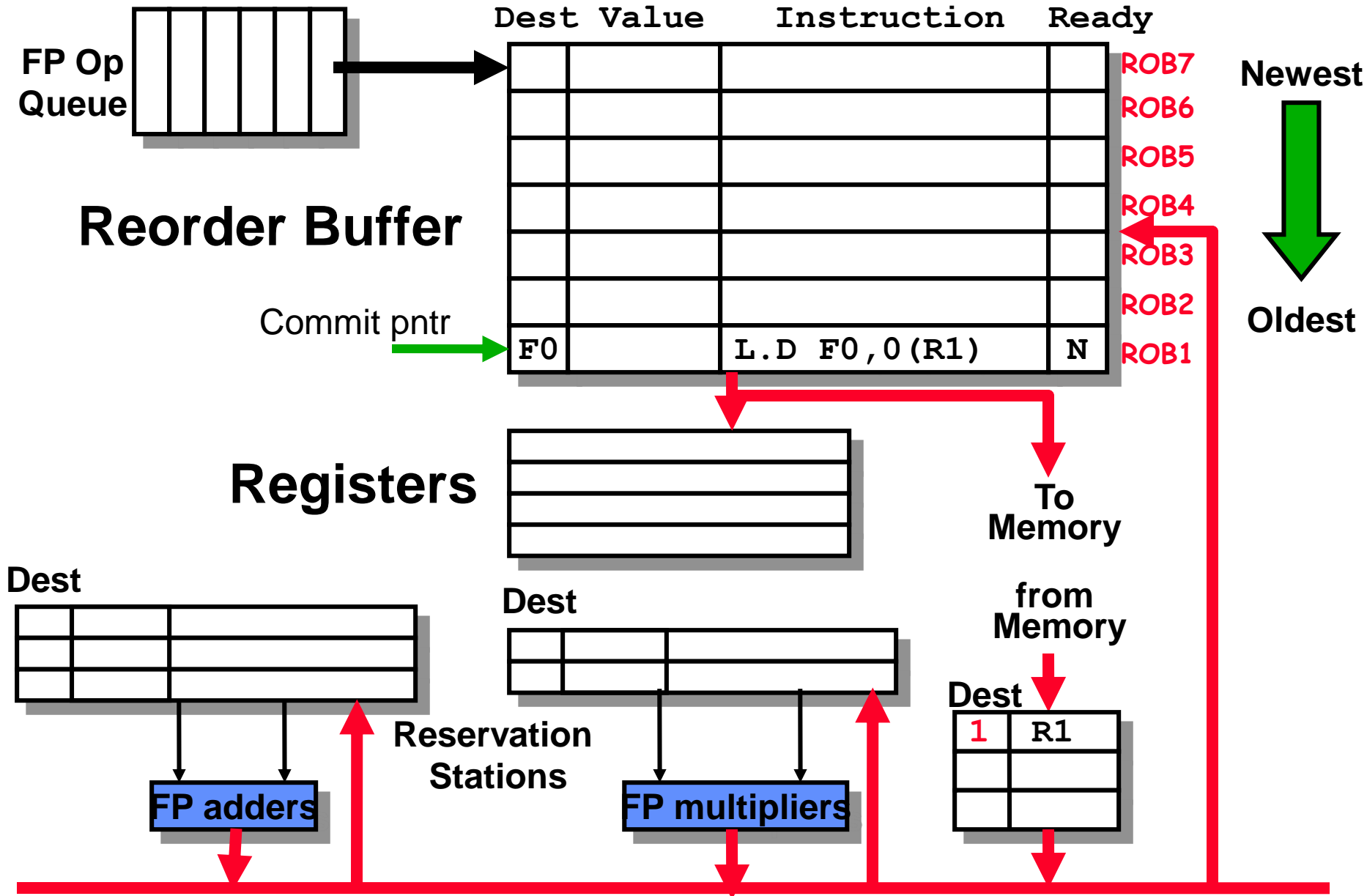
In-order commit

Status	Wait until	Action or bookkeeping
Issue all instructions	Reservation station (r) and ROB (b) both available	<pre> if (RegisterStat[rs].Busy) /*in-flight instr. writes rs*/ {h ← RegisterStat[rs].Reorder; if (ROB[h].Ready) /* Instr completed already */ {RS[r].Vj ← ROB[h].Value; RS[r].Qj ← 0;} else {RS[r].Qj ← h;} /* wait for instruction */ } else {RS[r].Vj ← Regs[rs]; RS[r].Qj ← 0;}; RS[r].Busy ← yes; RS[r].Dest ← b; ROB[b].Instruction ← opcode; ROB[b].Dest ← rd;ROB[b].Ready ← no; </pre>
FP operations and stores		<pre> if (RegisterStat[rt].Busy) /*in-flight instr writes rt*/ {h ← RegisterStat[rt].Reorder; if (ROB[h].Ready) /* Instr completed already */ {RS[r].Vk ← ROB[h].Value; RS[r].Qk ← 0;} else {RS[r].Qk ← h;} /* wait for instruction */ } else {RS[r].Vk ← Regs[rt]; RS[r].Qk ← 0;}; </pre>
FP operations		<pre> RegisterStat[rd].Reorder ← b; RegisterStat[rd].Busy ← yes; ROB[b].Dest ← rd; </pre>
Loads		<pre> RS[r].A ← imm; RegisterStat[rt].Reorder ← b; RegisterStat[rt].Busy ← yes; ROB[b].Dest ← rt; </pre>
Stores		<pre> RS[r].A ← imm; </pre>

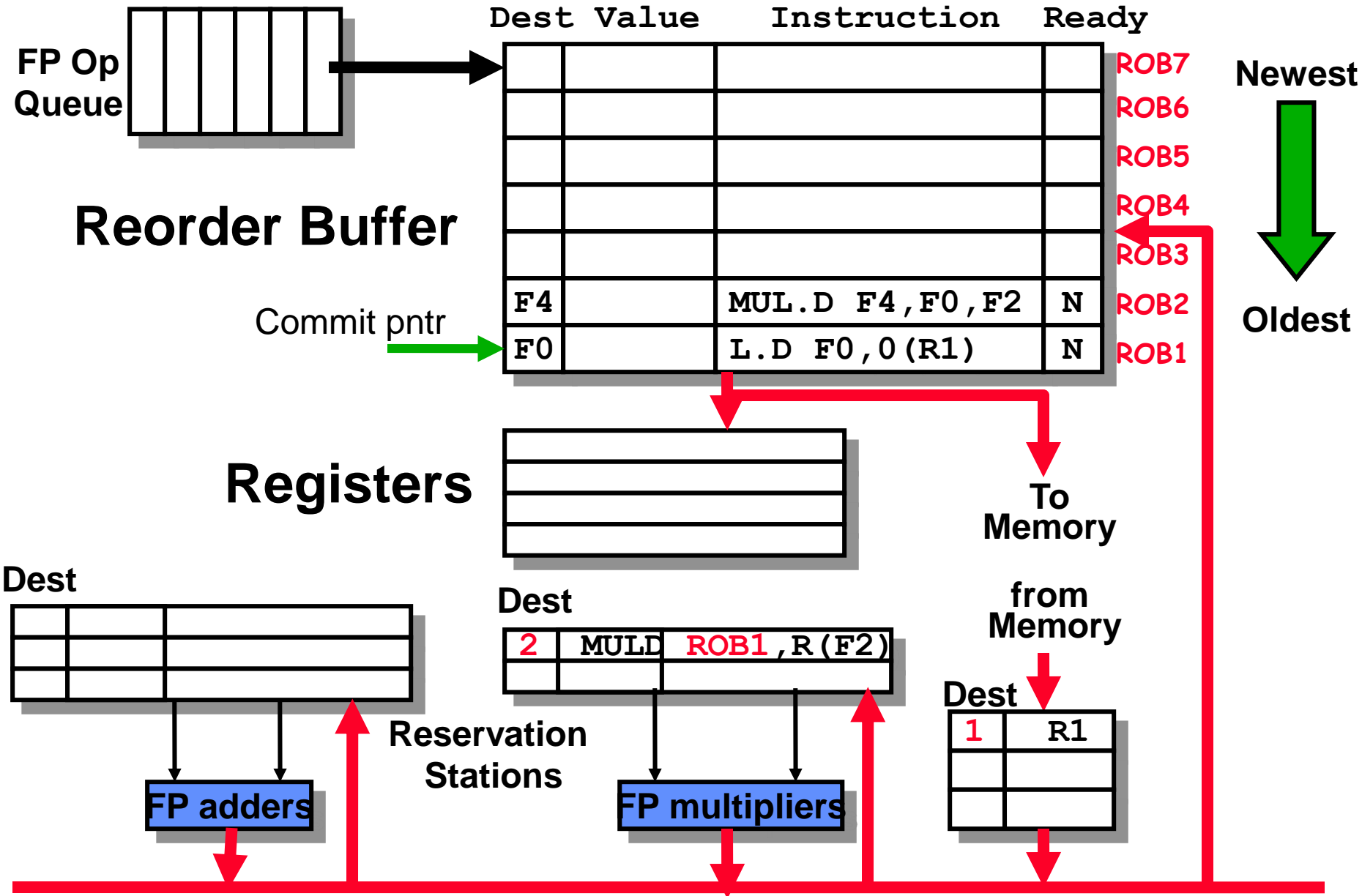
Status	Wait until	Action or bookkeeping
Execute FP op	$(RS[r].Q_j == 0)$ and $(RS[r].Q_k == 0)$	Compute results—operands are in V_j and V_k
Load step 1	$(RS[r].Q_j == 0)$ and there are no stores earlier in the queue	$RS[r].A \leftarrow RS[r].V_j + RS[r].A;$
Load step 2	Load step 1 done and all stores earlier in ROB have different address	Read from $Mem[RS[r].A]$
Store	$(RS[r].Q_j == 0)$ and store at queue head	$ROB[h].Address \leftarrow RS[r].V_j + RS[r].A;$

Status	Wait until	Action or bookkeeping
Write result all but store	Execution done at r and CDB available	$b \leftarrow RS[r].Dest; RS[r].Busy \leftarrow no;$ $\forall x (if (RS[x].Qj==b) \{RS[x].Vj \leftarrow result; RS[x].Qj \leftarrow 0\});$ $\forall x (if (RS[x].Qk==b) \{RS[x].Vk \leftarrow result; RS[x].Qk \leftarrow 0\});$ $ROB[b].Value \leftarrow result; ROB[b].Ready \leftarrow yes;$
Store	Execution done at r and $(RS[r].Qk == 0)$	$ROB[h].Value \leftarrow RS[r].Vk;$
Commit	Instruction is at the head of the ROB (entry h) and $ROB[h].ready == yes$	$d \leftarrow ROB[h].Dest; /* register dest, if exists */$ $if (ROB[h].Instruction==Branch)$ $\quad \{if (branch is mispredicted)$ $\quad \quad \{clear ROB[h], RegisterStat; fetch branch dest;\};\}$ $else if (ROB[h].Instruction==Store)$ $\quad \{Mem[ROB[h].Destination] \leftarrow ROB[h].Value;\}$ $else /* put the result in the register destination */$ $\quad \{Regs[d] \leftarrow ROB[h].Value;\};$ $ROB[h].Busy \leftarrow no; /* free up ROB entry */$ $/* free up dest register if no one else writing it */$ $if (RegisterStat[d].Reorder==h) \{RegisterStat[d].Busy \leftarrow no;\};$

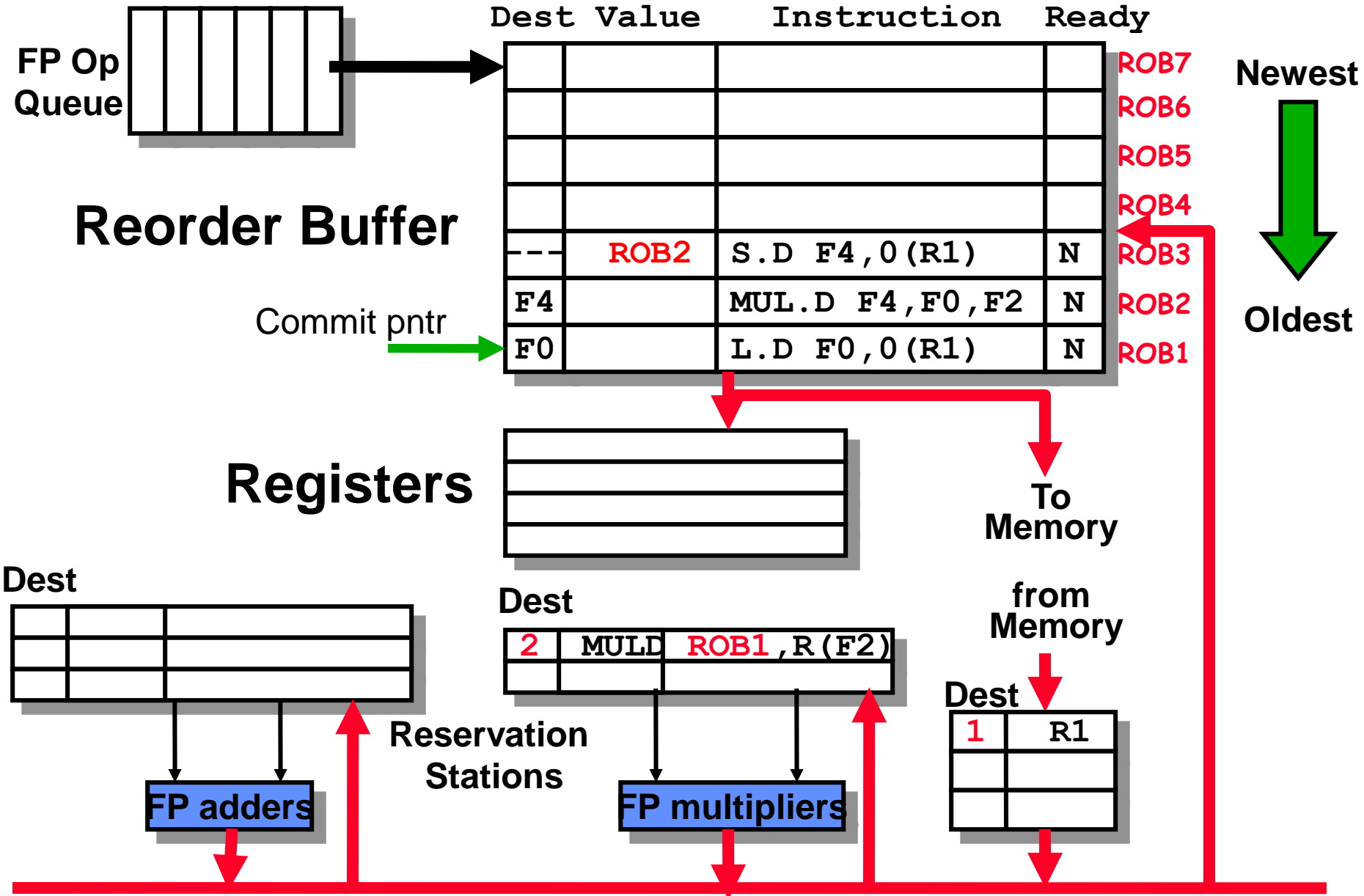
Tomasulo With Reorder buffer(1)



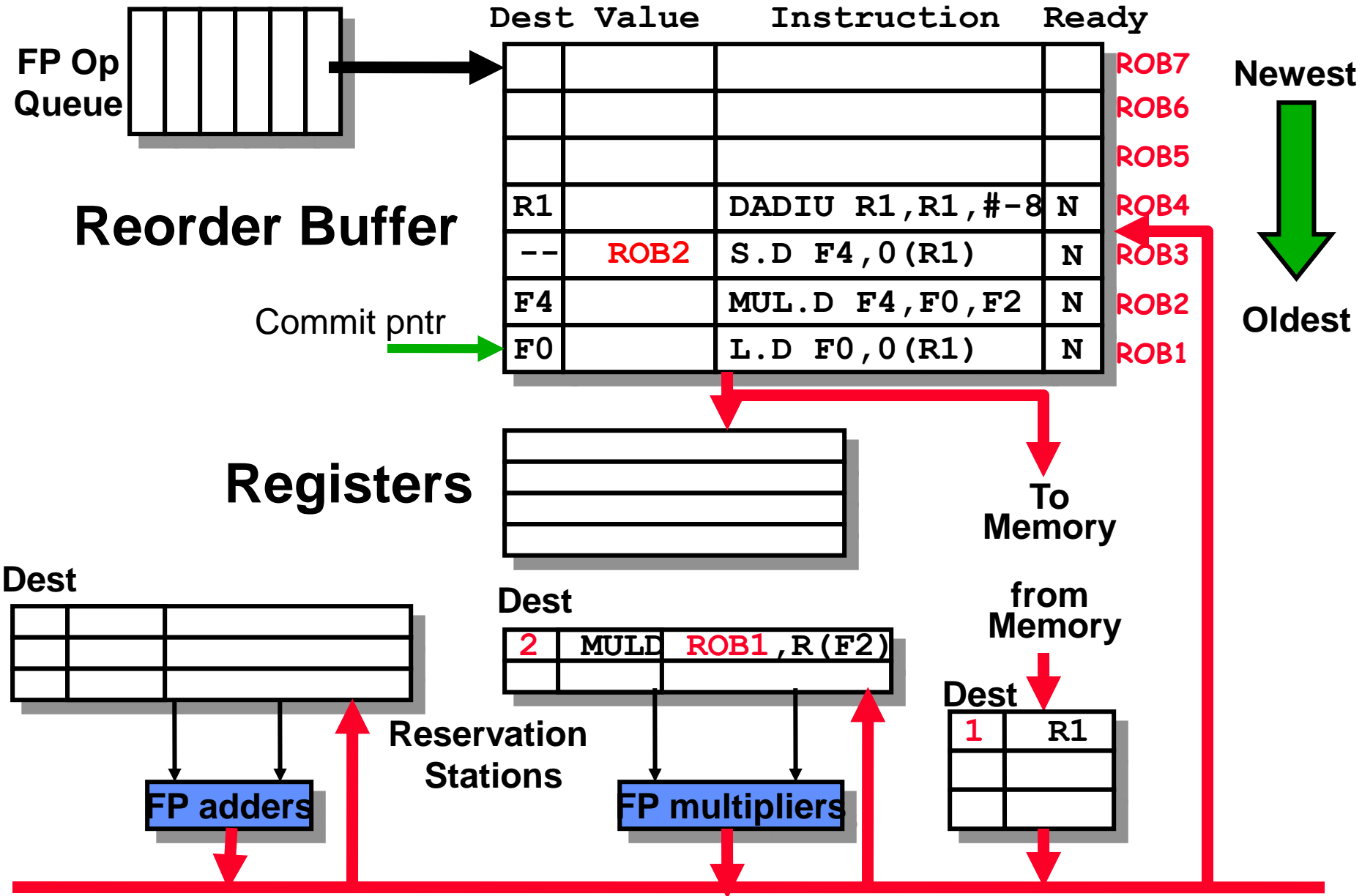
Tomasulo With Reorder buffer(2)



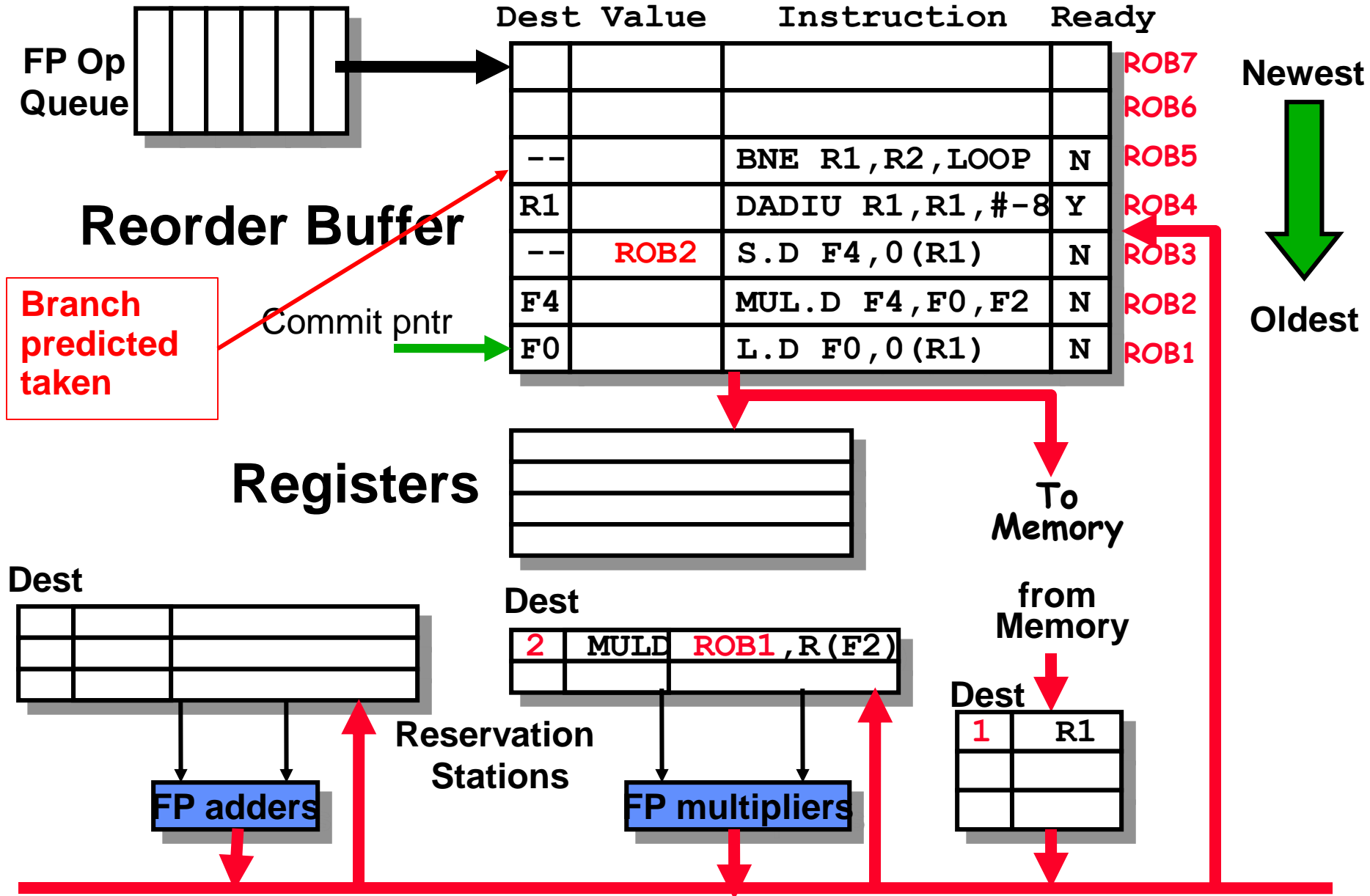
Tomasulo With Reorder buffer(3)



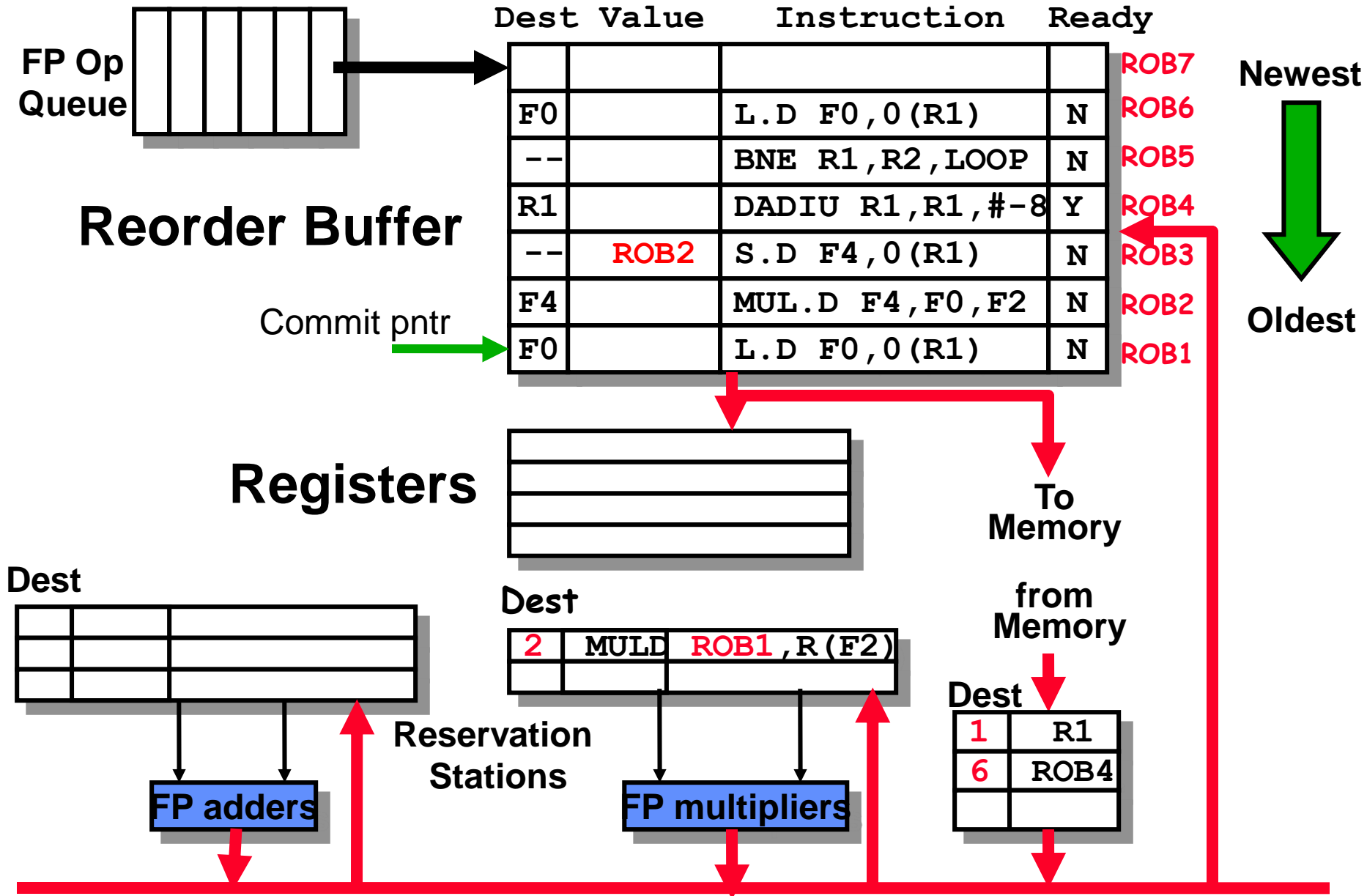
Tomasulo With Reorder buffer(4)



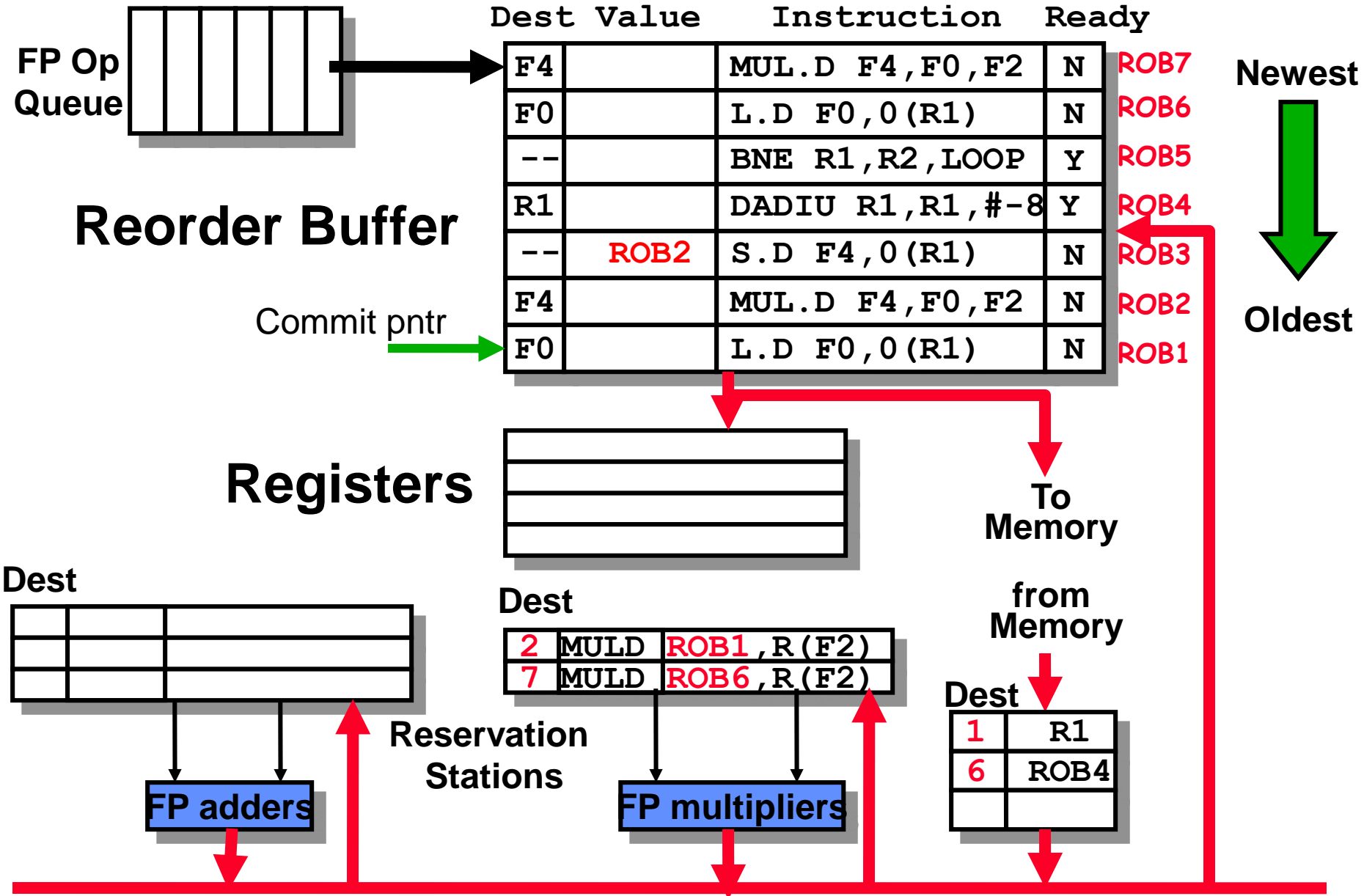
Tomasulo With Reorder buffer(5)



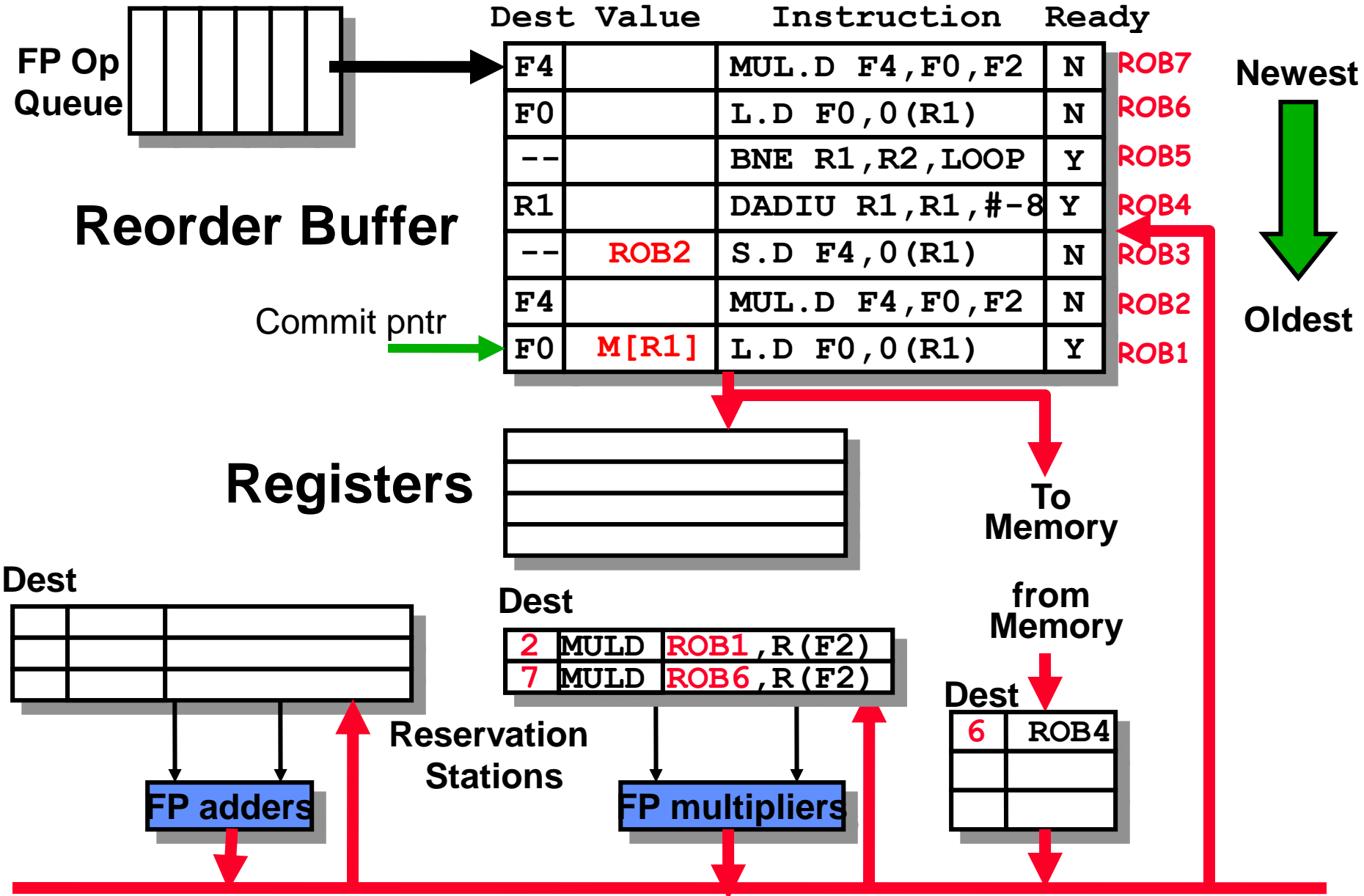
Tomasulo With Reorder buffer(6)



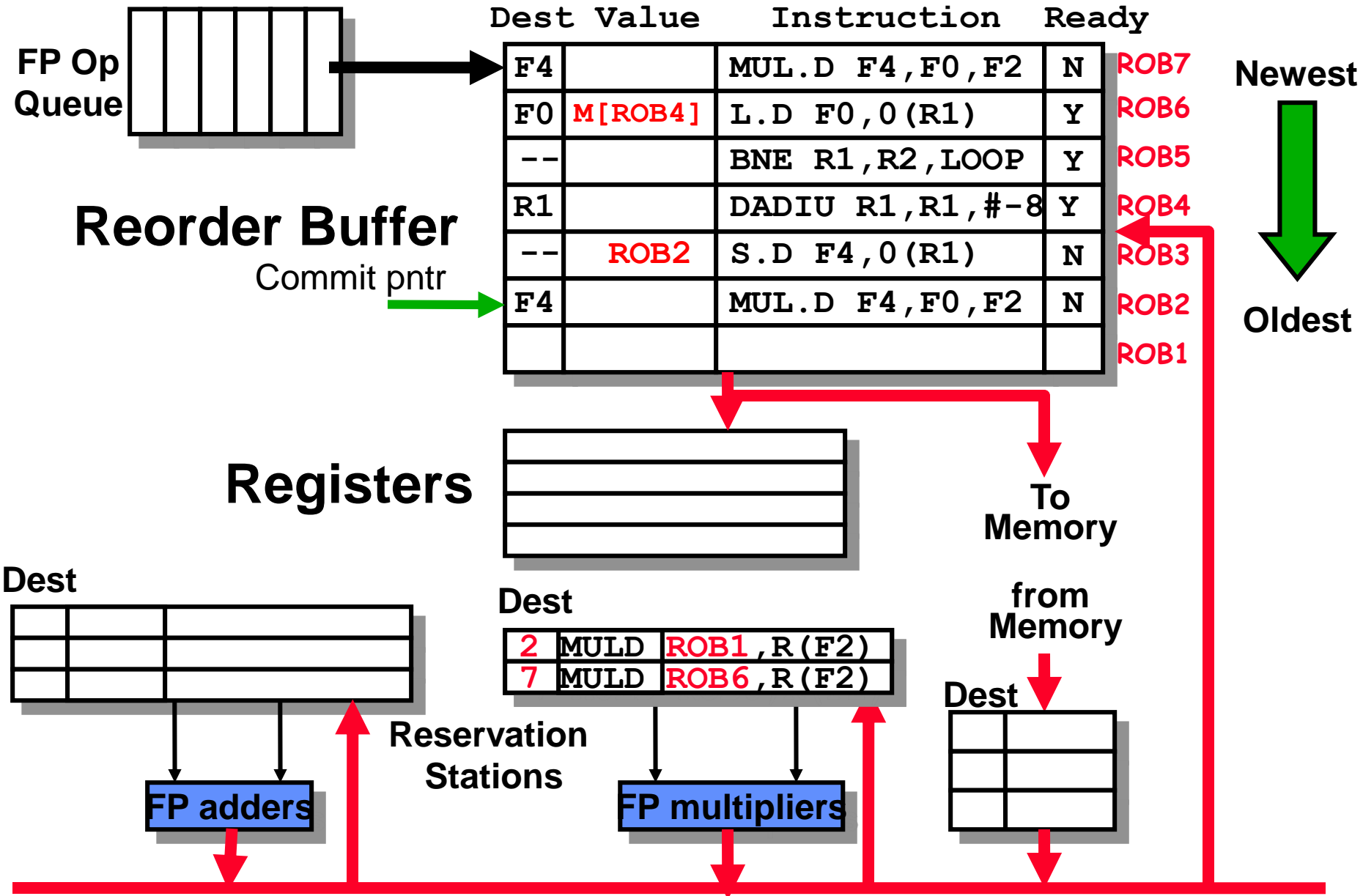
Tomasulo With Reorder buffer(7)



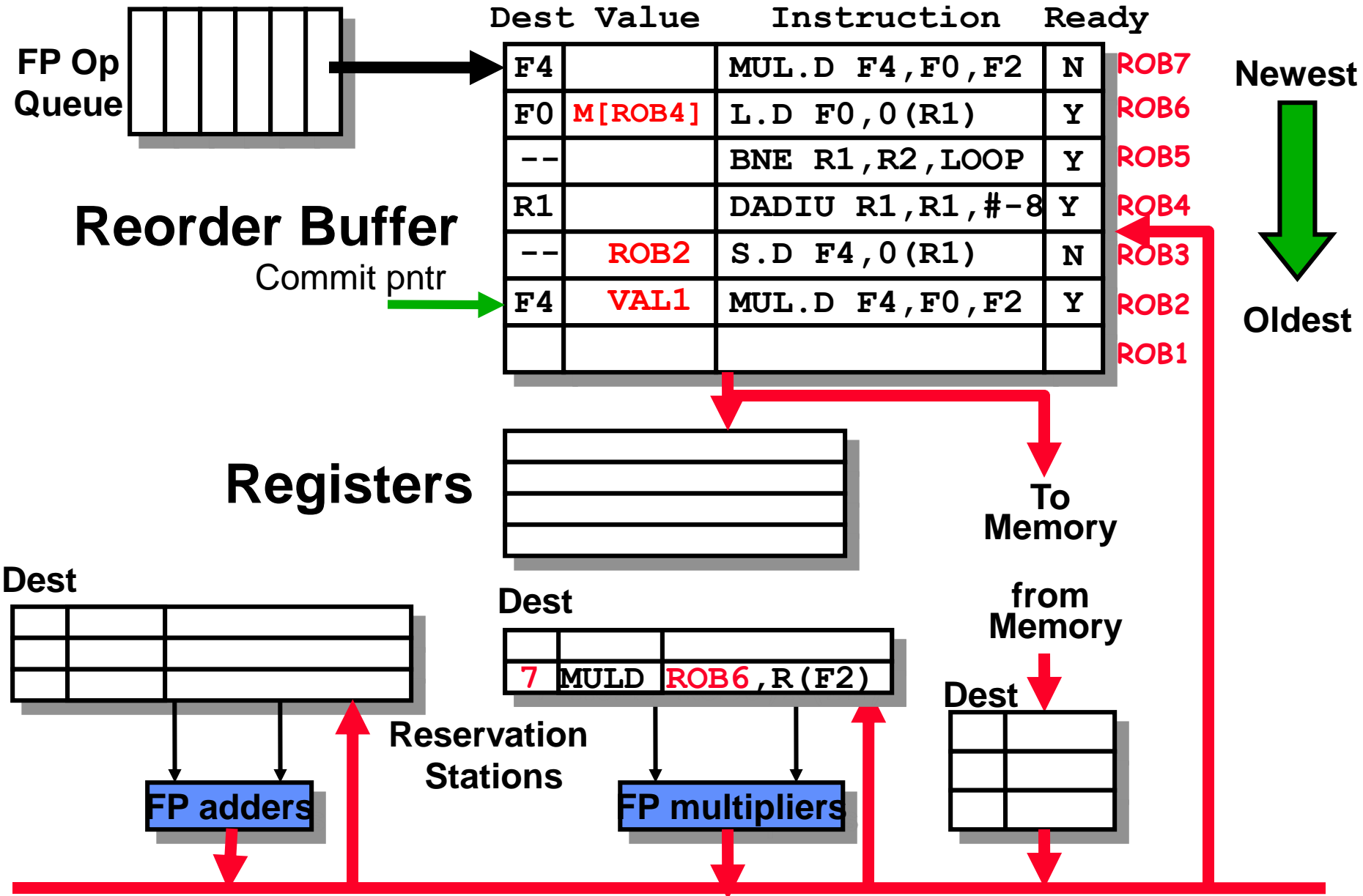
Tomasulo With Reorder buffer(8)



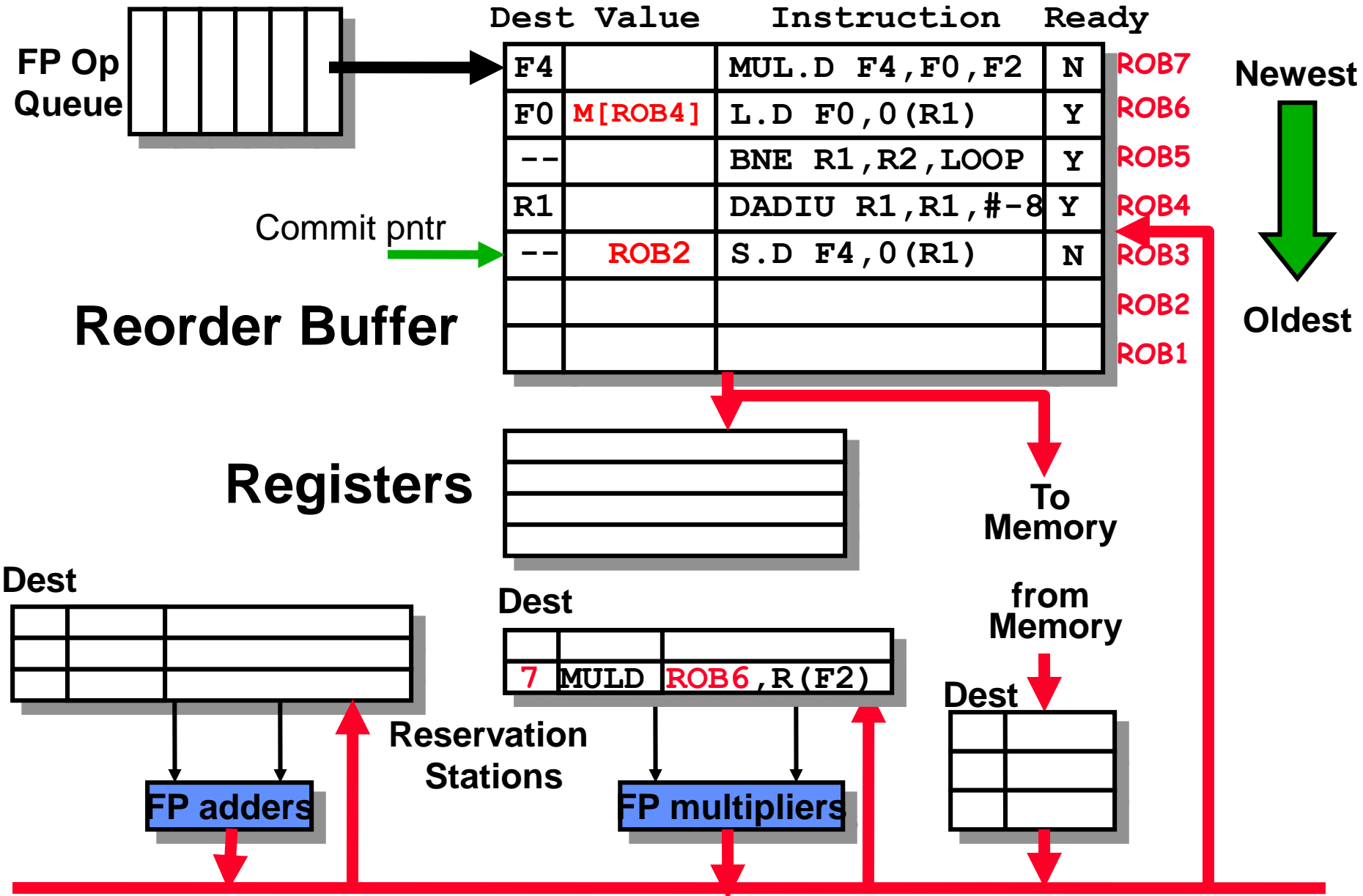
Tomasulo With Reorder buffer(9)



Tomasulo With Reorder buffer(12)



Tomasulo With Reorder buffer(13)



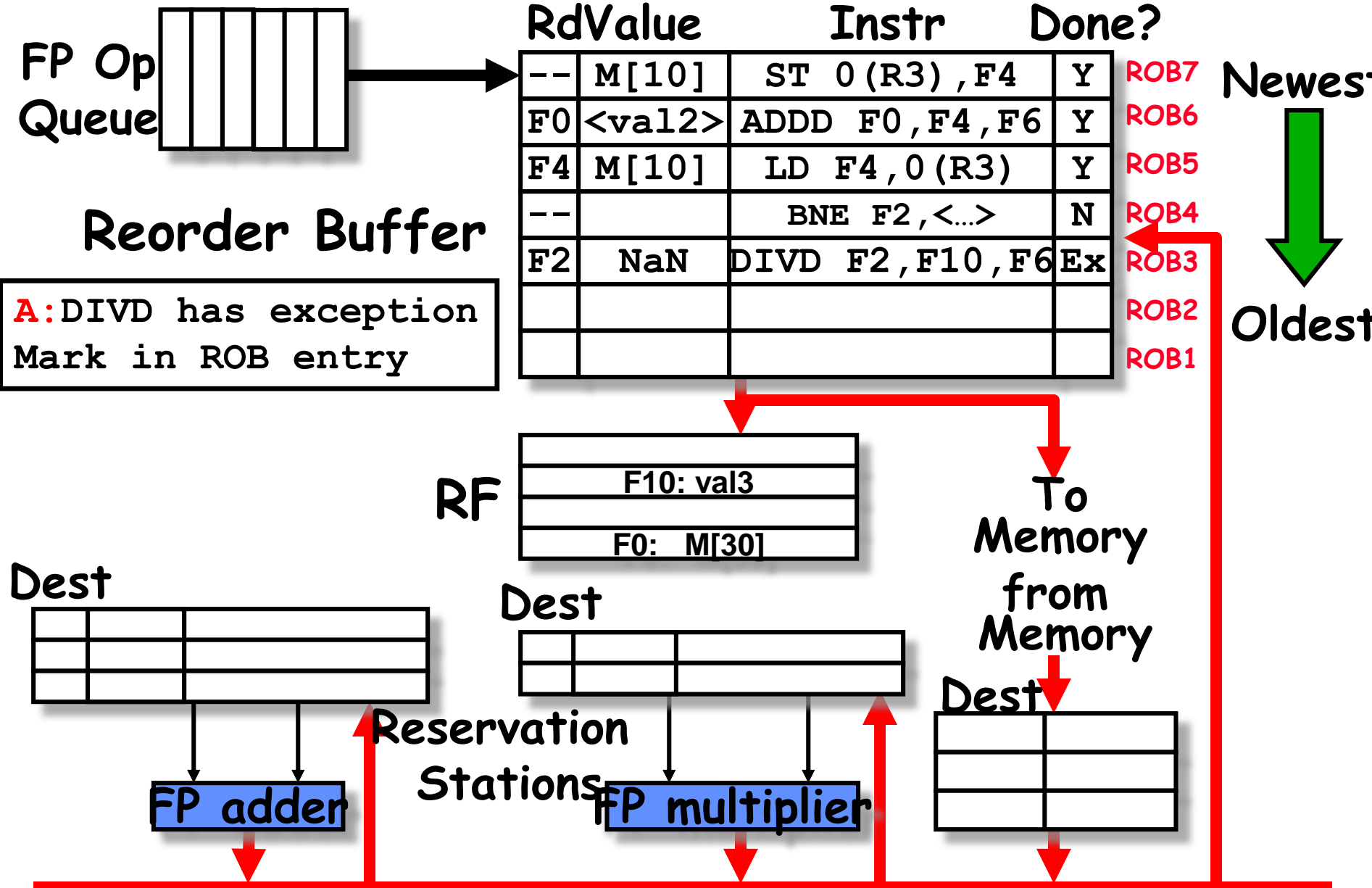
Tomasulo With Reorder buffer(13)

- Στο σημείο αυτό γίνεται commit το SD, ενώ καθώς αδειάζει ο ROB εισέρχονται καινούριες εντολές.
- Οι άλλες εντολές (DADDIU,BNE,LD) ενώ έχουν εκτελεστεί **δεν έχουν γίνει ακόμα commit.**
- Αν η πρόβλεψη **taken** για το BNE ήταν **σωστή**, οι επόμενες εντολές που έχουν εκτελεστεί γίνονται κανονικά commit.
- Αν ανακαλύψουμε όμως ότι ήταν **λάθος**, τότε απλά διαγράφουμε τις εντολές από το BNE και πάνω μέσα στον ROB. Η εκτέλεση συνεχίζεται από το σωστό σημείο.
 - Aggressive συστήματα θα μπορούσαν να κάνουν τη διαγραφή πριν ο commit_ptr φτάσει στο BNE

Exceptions and Interrupts

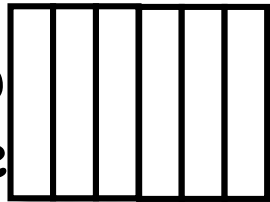
- IBM 360/91 invented “imprecise interrupts”
 - Computer stopped at this PC; its likely close to this address
 - Not so popular with programmers
 - Also, what about Virtual Memory? (Not in IBM 360)
- Technique for both precise interrupts/exceptions and speculation: in-order completion and in-order commit
 - If we speculate and are wrong, need to back up and restart execution to point at which we predicted incorrectly
 - This is exactly same as need to do with precise exceptions
- Exceptions are handled by not recognizing the exception until instruction that caused it is ready to commit in ROB
 - If a speculated instruction raises an exception, the exception is recorded in the ROB
 - This is why reorder buffers in all new processors

Exceptions With Reorder Buffer:



Exceptions With Reorder Buffer:

FP Op Queue



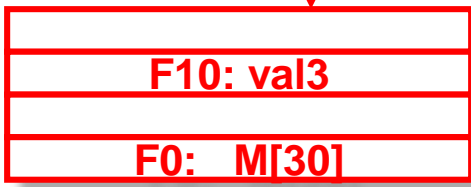
Reorder Buffer

A: DIVD commits
 → Exception is taken!!!
 LD, ADDD, ST killed
 Need: Status, PC

RdValue	Instr	Done?
-- M[10]	ST 0 (R3), F4	Y
F0 <val2>	ADDD F0, F4, F6	Y
F4 M[10]	LD F4, 0 (R3)	Y
--	BNE F2, <...>	N
F2 NaN	DIVD F2, F10, F6	Ex

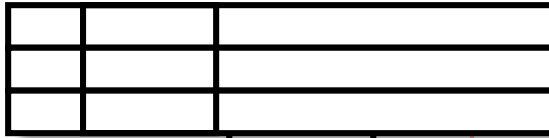
ROB7
ROB6
ROB5
ROB4
ROB3
ROB2
ROB1

Newest
↓
Oldest



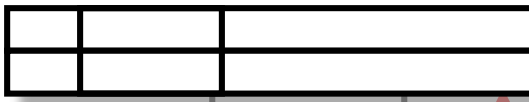
To Memory
from Memory

Dest



FP adder

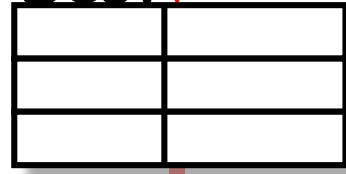
Dest



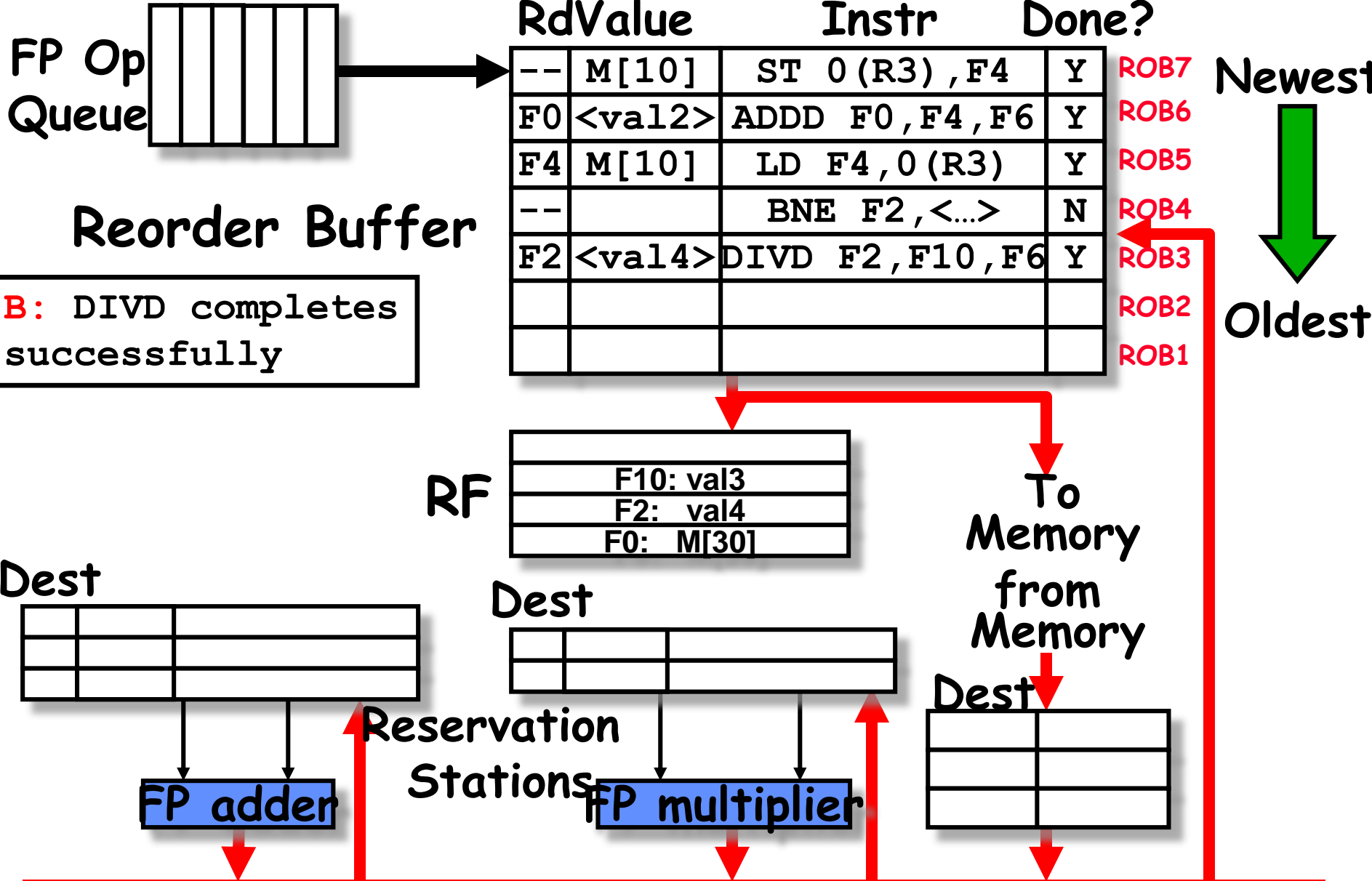
FP multiplier

Reservation Stations

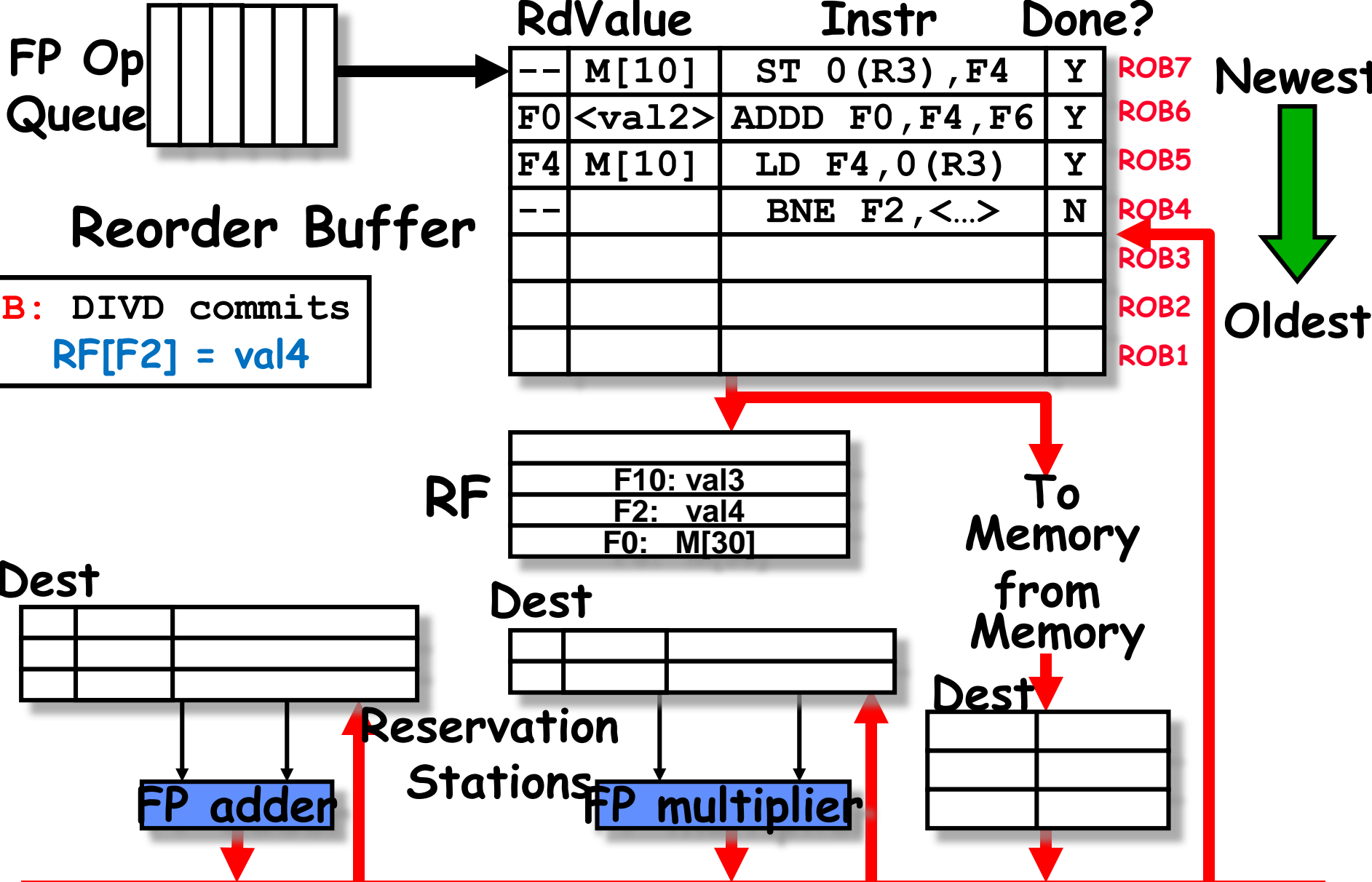
Dest



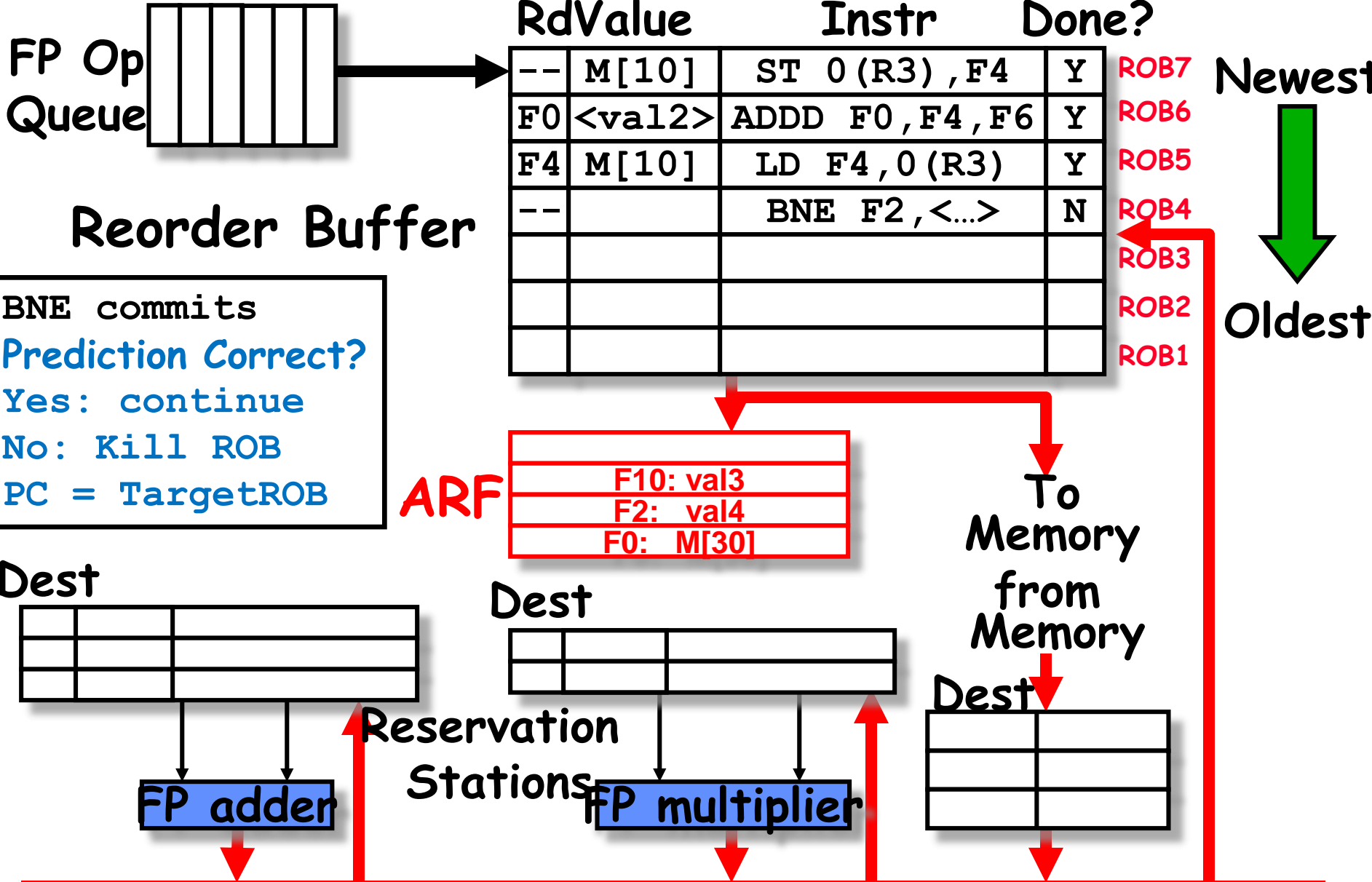
Tomasulo With Reorder Buffer:



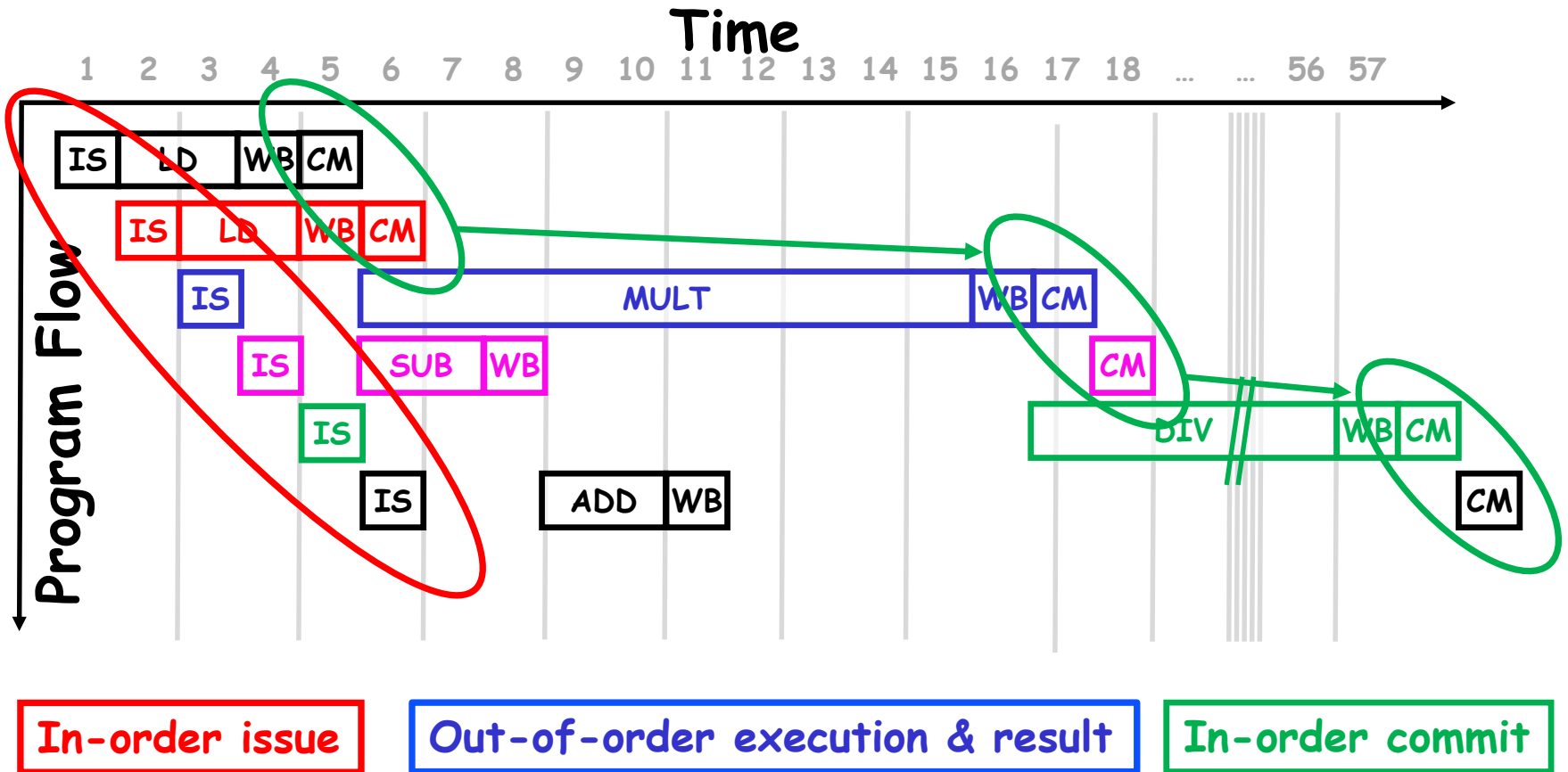
Tomasulo With Reorder Buffer:



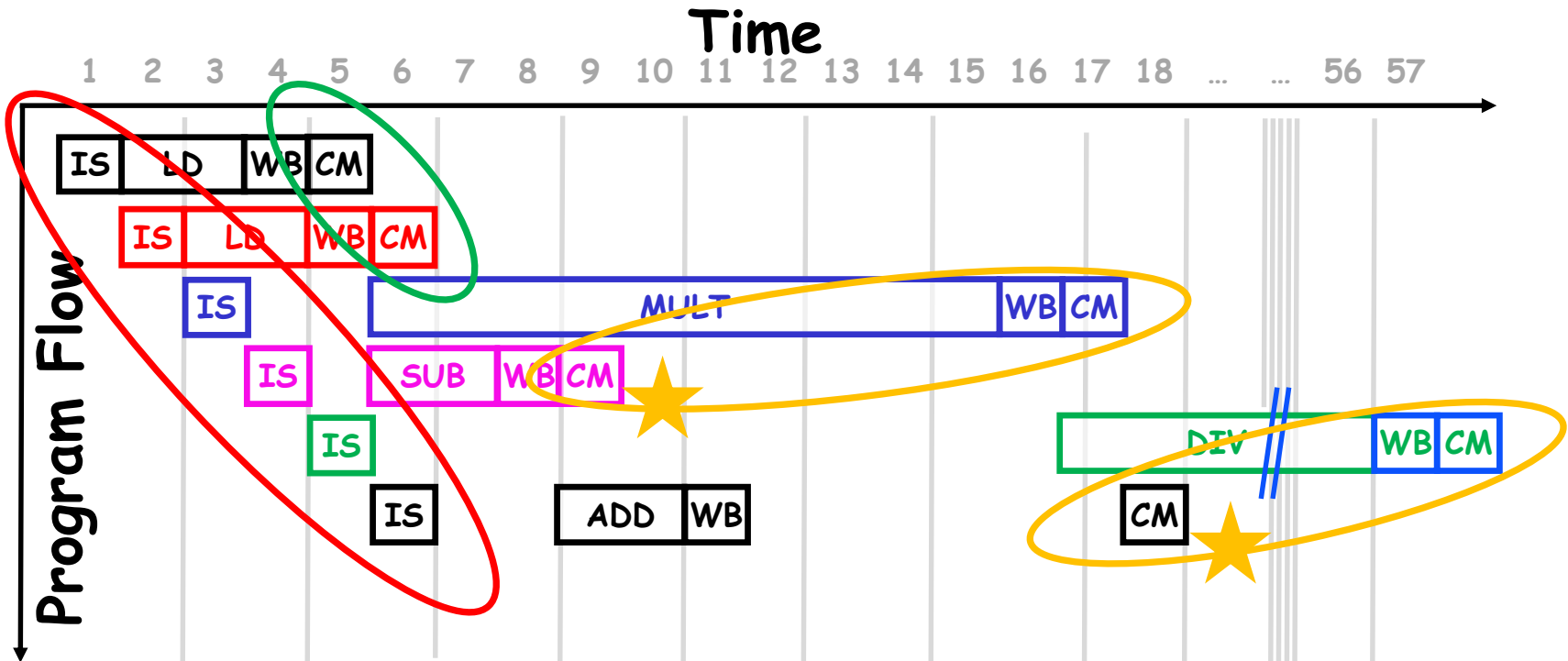
Tomasulo With Reorder Buffer:



When to commit?



When to commit?



Can this be legal?

No: Easy answer, no thinking

Yes: But only if:

- Can guarantee no exceptions from before
- Cannot stop after DIV, only after ADD
- Coarse valid commit boundaries

Is this Interesting/Useful?

Precise Points

- ISA: each instruction defines a “clean” point to stop
- BUT, why do we need to stop/restart?
 - External Interrupts: we can choose *if* and *when* to stop
 - » Maskable interrupts, priorities, etc.
 - Exceptions:
 - » Internal to the program, cannot choose if and when!
 - » Non-restartable: fatal, kill program, all is lost, easy!
 - » Restartable: Which ones? Not all instructions!!!
- Fine, we have to restart execution
 - What if we restart from an earlier point and re-execute more instructions? Is it valid? Is the cost acceptable?
- Commit points:
 - Branch
 - Ld/St
- Perhaps branch only? (remember basic blocks?)

Precise Points

Needed restart points in the following program?

Label:

LD	F6	34+	R2	✓ address/perm
MULTD		F0	F2	F4 ✓ overflow
LD	F2	45+	R3	✓ address/prem
SUBD		F8	F6	F2 ✓ overflow
subi		R2	R2	1
bne	R2	R8	Label	✓ misprediction

Assume exception in second LD. Can I re-execute *all* instructions (from Label)?

- Yes, if I have the ARF+PC+Mem for Label

Coarse-grain Precise Points

- Keep enough info only for *some* points in the program
Branch (and perhaps mem) instructions
- On “problem” (exception/misprediction/etc)
 - Return to the latest safe point before problem i.e. undo/kill all instructions after that point, restart from PC
 - All memory effects should be undoable:
Writes should not reach memory until committed

Will return to this when we study current processor structure & mechanisms

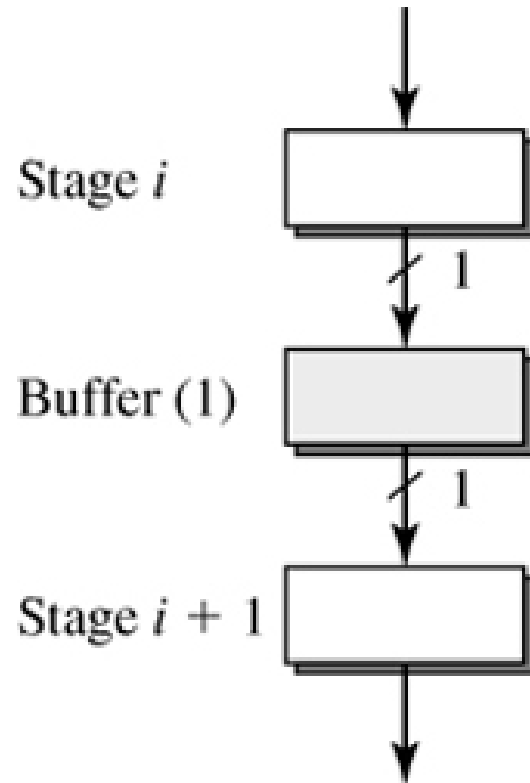
Hazards

- Δεν υπάρχουν **WAW**, **WAR** hazards
- Για την αποφυγή **RAW** hazards ακολουθούμε τα εξής βήματα:
 - Αν κάποιο store προηγείται ενός load, καθυστερούμε το load μέχρι το store να υπολογίσει το address του
 - Αν $\text{load_address} = \text{store_address}$, τότε υπάρχει πιθανότητα RAW και :
 - Αν η τιμή του store **είναι** γνωστή, την περνάμε στο load
 - Αν η τιμή του store **δεν είναι** γνωστή, τότε το load χρησιμοποιεί ως source το νούμερο του ROB που περιέχει το store
 - Αποστολή του request στη μνήμη

Out-of-order εκτέλεση: επανεξέταση υπερβαθμωτών αρχιτεκτονικών...

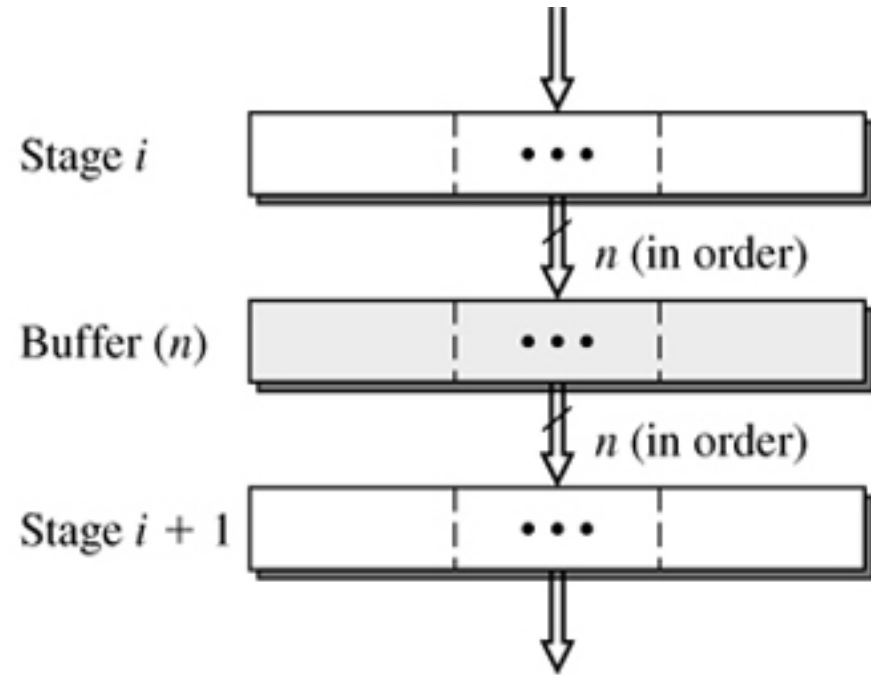
Βαθμωτή αρχιτεκτονική αγωγού

- μεταξύ δύο σταδίων του pipeline υπάρχει πάντα ένας ενδιάμεσος καταχωρητής δεδομένων (pipeline register ή buffer)
- στις βαθμωτές σωληνώσεις, κάθε buffer έχει μία μόνο θύρα εισόδου και μία εξόδου



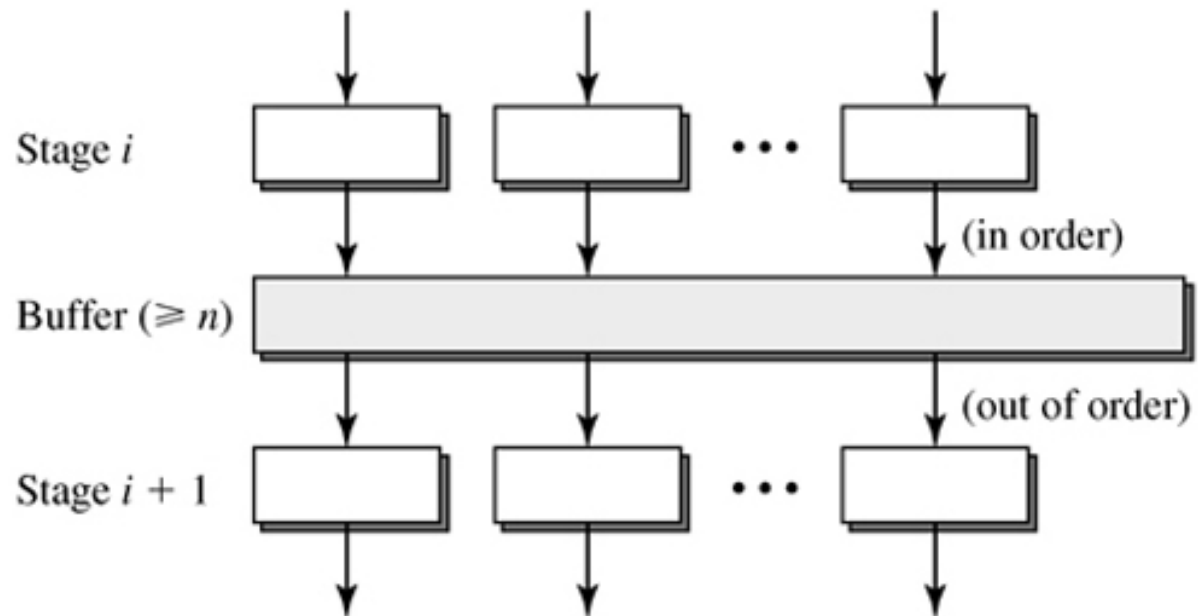
Υπερβαθμωτή αρχιτεκτονική αγωγού

- κάθε buffer έχει πολλαπλές θύρες εισόδου και εξόδου, και αποτελείται από πολλαπλούς ενδιάμεσους καταχωρητές
- κάθε ενδιάμεσος καταχωρητής μπορεί να λάβει δεδομένα μόνο από μία θύρα
- δεν υπάρχει δυνατότητα διακίνησης των δεδομένων μεταξύ των καταχωρητών του ίδιου buffer



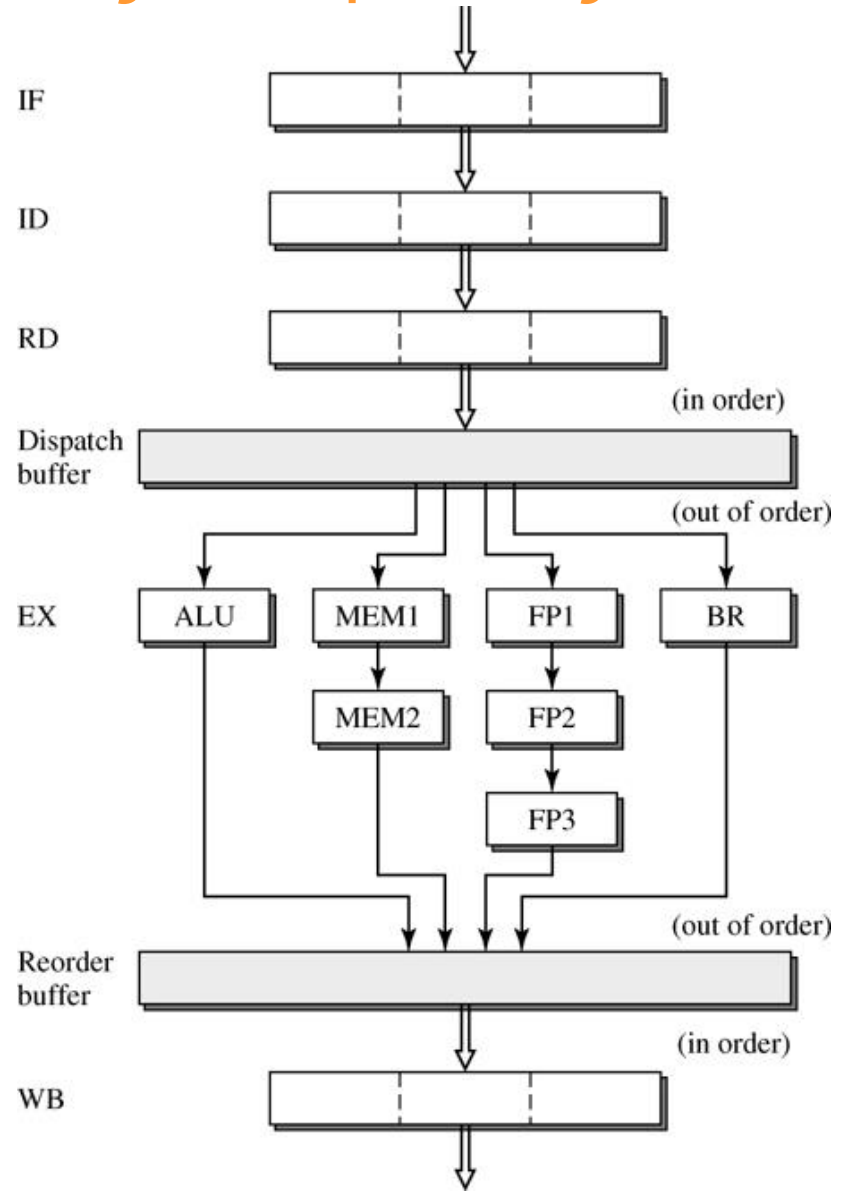
Υπερβαθμωτή αρχιτεκτονική αγωγού – εκτέλεση εκτός σειράς (out-of-order)

- οι εντολές που καθυστερούν μέσα στη σωλήνωση (stall) παρακάμπτονται και συνεχίζεται η ροή εκτέλεσης με τις επόμενες εντολές
- out-of-order execution ή dynamic scheduling



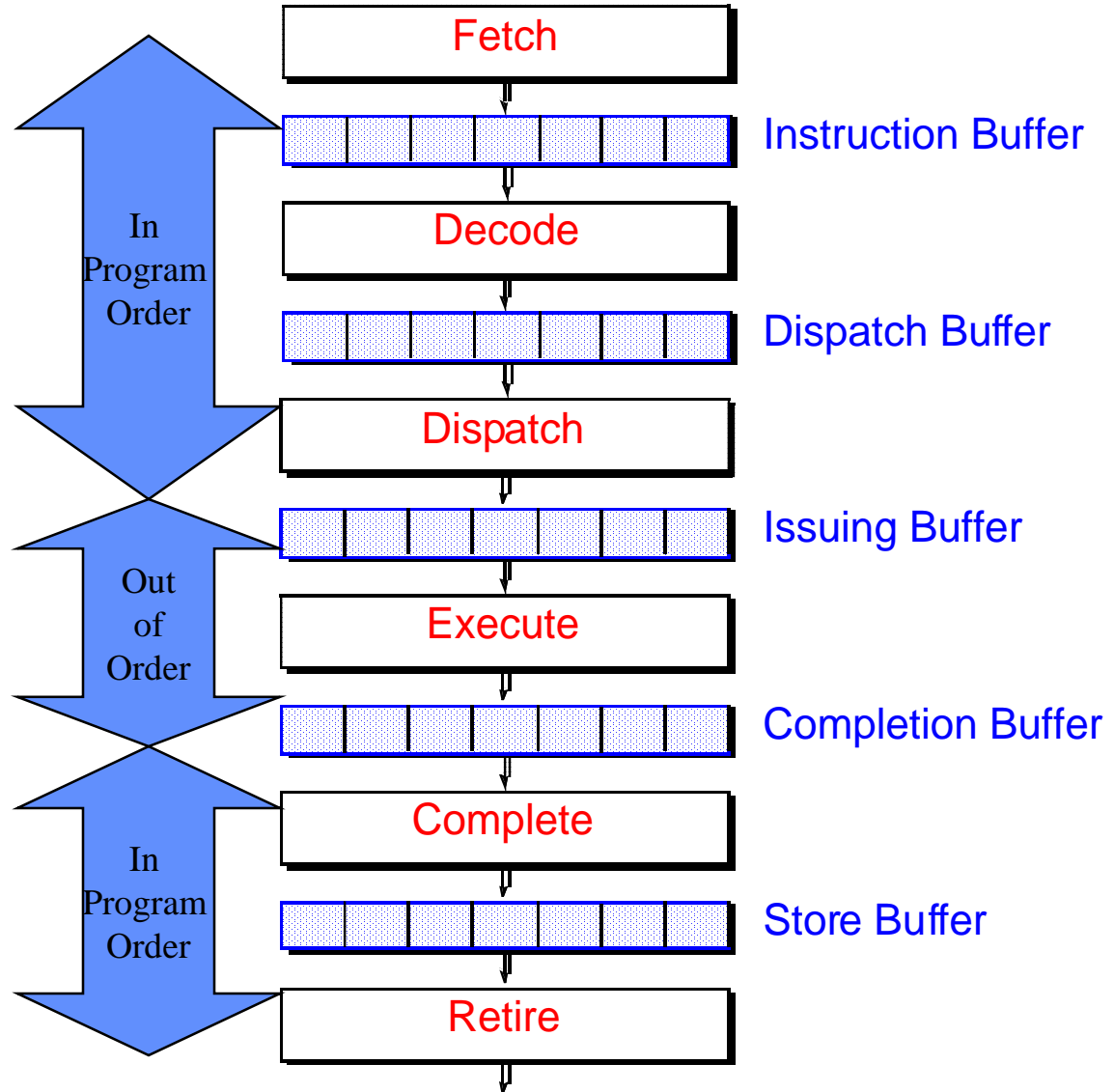
Δυναμική αρχιτεκτονική αγωγού, με πολλαπλές, ετερογενείς σωληνώσεις

- ο dispatch buffer λαμβάνει τις εντολές εν σειρά, και τις κατανέμει στις λειτουργικές μονάδες εκτός σειράς
- όταν οι εντολές ολοκληρώσουν την εκτέλεσή τους, ο reorder buffer τις αναδιατάσσει, σύμφωνα με τη σειρά που υπαγορεύει το πρόγραμμα, προκειμένου να πιστοποιήσει τη σωστή ολοκλήρωσή τους



Γενική μορφή υπερβαθμωτής αρχιτεκτονικής αγωγού

- το στάδιο εκτέλεσης (execute) μπορεί να περιλαμβάνει πολλαπλές διαφορετικού τύπου σωληνώσεις, με διαφορετικό latency η κάθε μία
- αναγκαία τα στάδια dispatch και complete, για την αναδιάταξη και επαναφορά των εντολών σε σειρά
- παρεμβολή ενδιάμεσων buffers ανάμεσα σε διαδοχικά στάδια

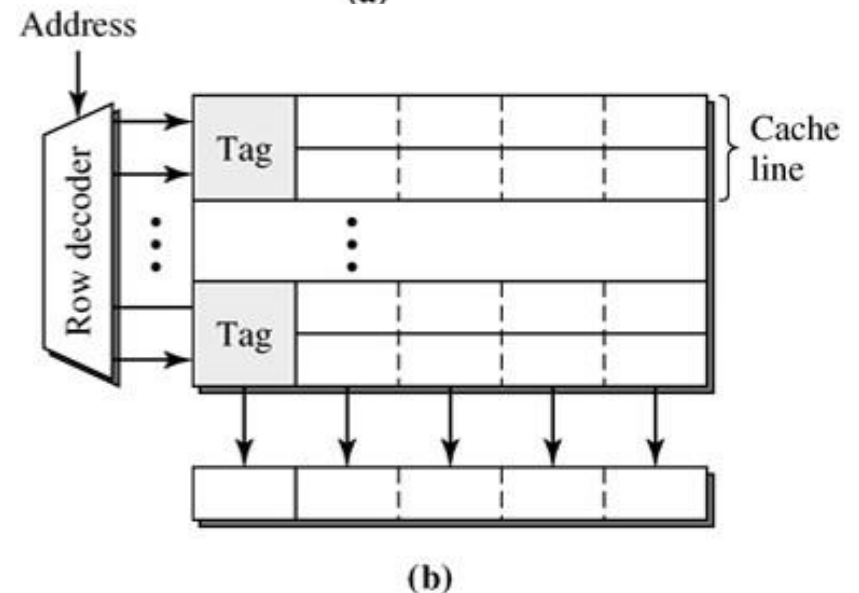
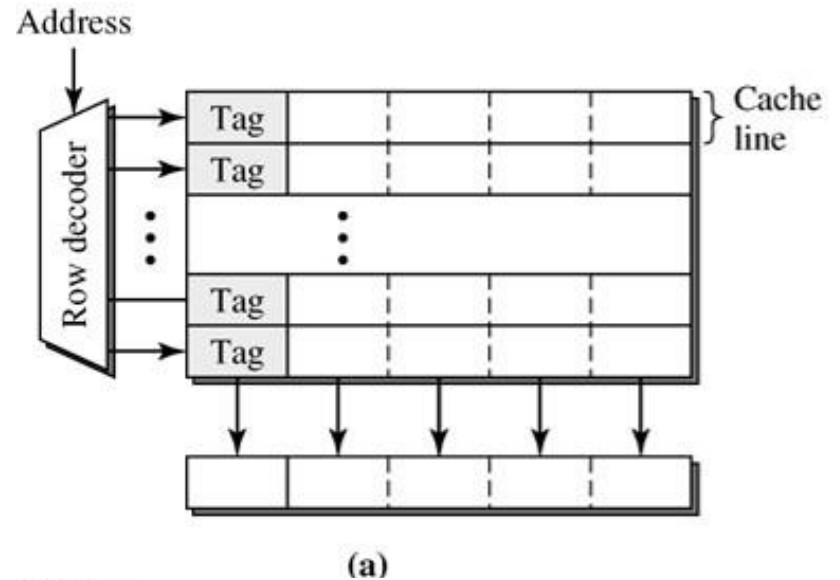


Superscalar Pipeline Design

- Instruction Fetching Issues
- Instruction Decoding Issues
- Instruction Dispatching Issues
- Instruction Execution Issues
- Instruction Completion & Retiring Issues

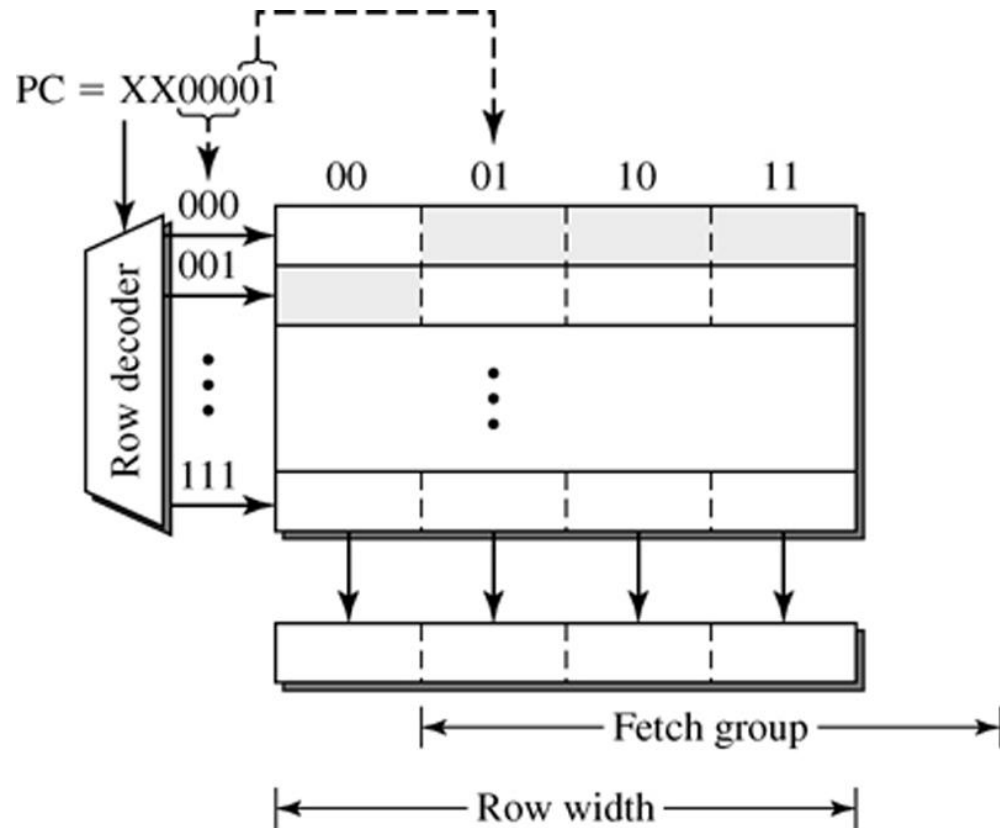
Ανάγνωση εντολής (Instruction Fetch)

- Σε μία αρχιτεκτονική πλάτους s , πρέπει διαβάζονται s εντολές σε κάθε κύκλο μηχανής από την κρυφή μνήμη εντολών (I-cache)
- Σε κάθε προσπέλαση της I-cache διαβάζεται μία ολόκληρη γραμμή (s εντολές ανά γραμμή): 1 cache line ανά γραμμή της I-cache (a). Θα μπορούσε μία cache line να επεκτείνεται σε περισσότερες της μίας γραμμής (b).



Ανάγνωση εντολής (Instruction Fetch)

- Για την ανάγνωση s εντολών σε κάθε κύκλο μηχανής θα πρέπει:
 - Οι s εντολές (που ανήκουν στο ίδιο **fetch group**) να είναι ευθυγραμμισμένες (aligned) στην I-cache



Ανάγνωση εντολής (Instruction Fetch)

- Εναλλακτικά, υπάρχουν μηχανισμοί ελέγχου της ροής των εντολών, που μπορούν να φροντίσουν για την ευθυγράμμιση των εντολών που ανήκουν στο ίδιο fetch group
 - Software (compiler static alignment)
 - Hardware (run time technique)

Αποκωδικοποίηση εντολών (Instruction Decode)

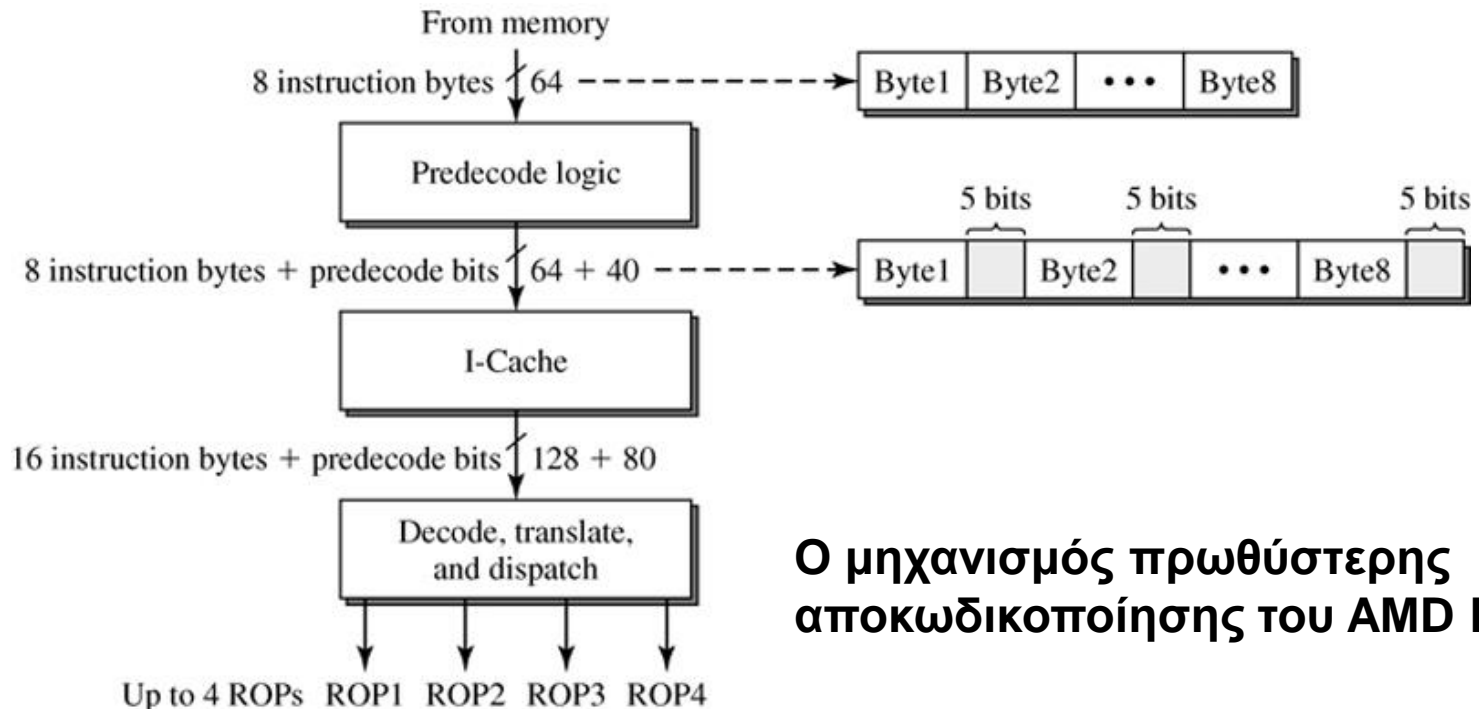
- Απλό για scalar RISC pipeline.
- Αναγνώριση των όποιων εξαρτήσεων μεταξύ εντολών – προώθηση των μη εξαρτημένων εντολών στο επόμενο στάδιο
- Αναγνώριση των εντολών άλματος
- Αποτελεί το κρισιμότερο στάδιο για την επίδοση ολόκληρης της αρχιτεκτονικής αγωγού
- Ακόμη μεγαλύτερη δυσκολία στις CISC εντολές: κάθε εντολή μεταφράζεται σε περισσότερες μικρο-εντολές (1,5-2 μops / CISC εντολή για την Intel)

Αποκωδικοποίηση εντολών (Instruction Decode)

- Η αποκωδικοποίηση των εντολών CISC σε παράλληλες αρχιτεκτονικές αγωγού ή οι πολύ πλατιές σωληνώσεις
 - Απαιτεί πολλαπλά στάδια αγωγού
 - Αυξάνει το branch penalty
- *Πώς μπορεί να αποφευχθεί η αύξηση του βάθους του τμήματος αποκωδικοποίησης στη σωλήνωση;*

Αποκωδικοποίηση εντολών (Instruction Decode)

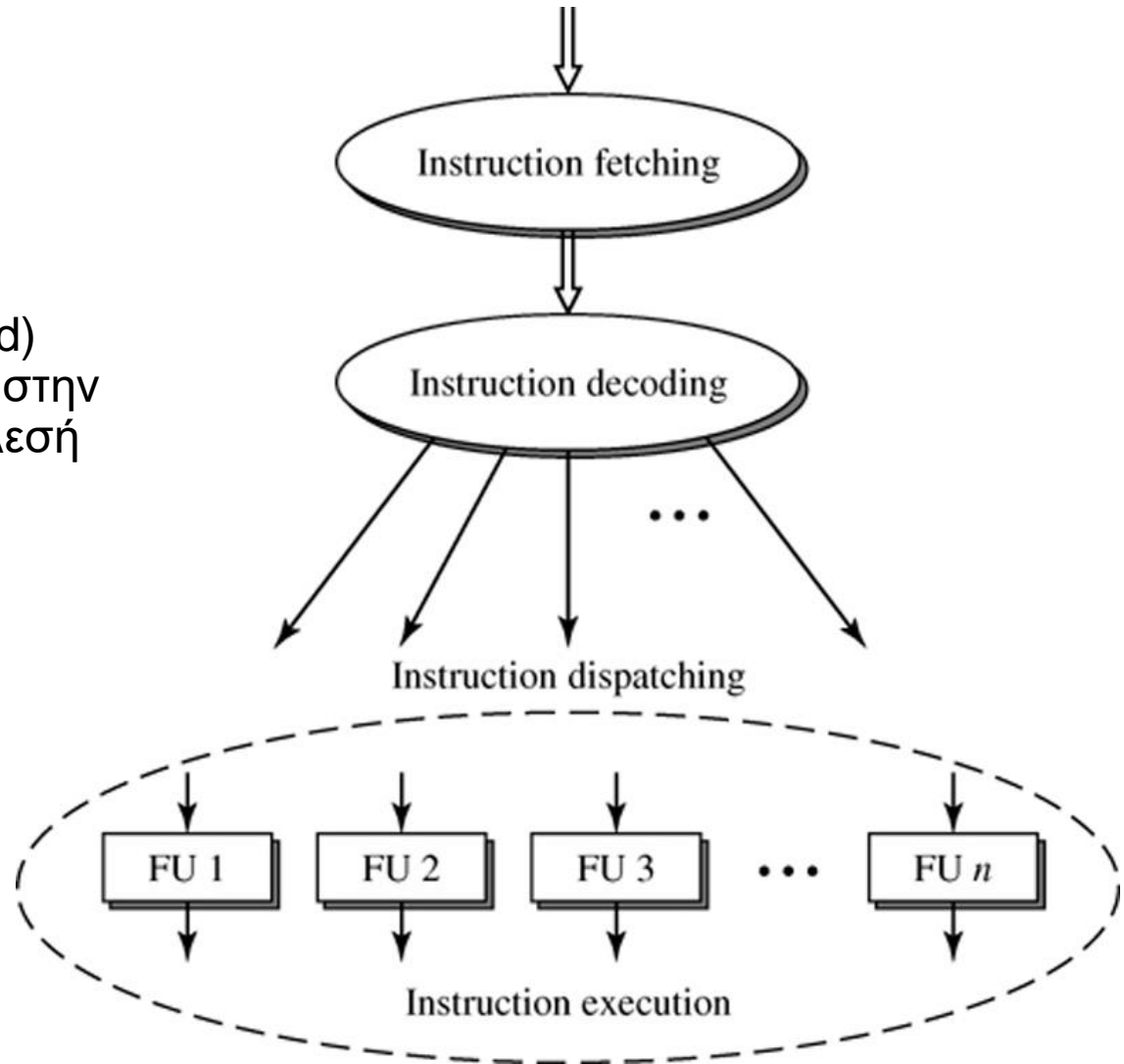
- Λύση:
 - Μερική αποκωδικοποίηση των εντολών πριν την είσοδό τους στην I-cache (**predecoding**)
- Αύξηση του I-cache miss penalty
- Αύξηση του μεγέθους της I-cache (για να συμπεριληφθούν τα predecoded bits)



Ο μηχανισμός πρωθύστερης αποκωδικοποίησης του AMD K5

Διανομή των εντολών – Instruction Dispatch

- Το στάδιο αυτό αναλαμβάνει τη μεταγωγή από την κεντρική (centralized) διαχείριση εντολών στην κατακεντρωμένη εκτέλεσή τους

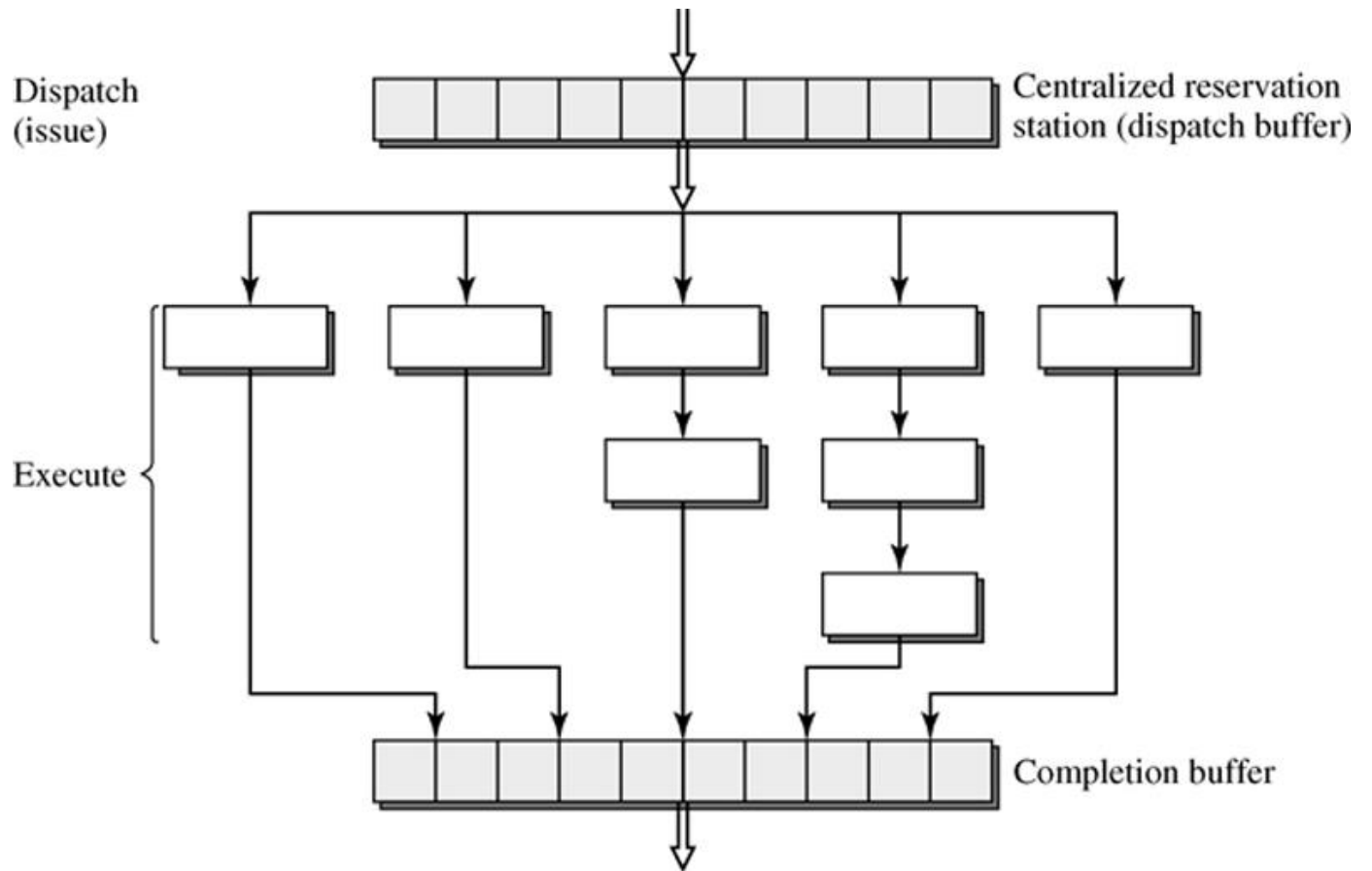


Διανομή των εντολών – Instruction Dispatch

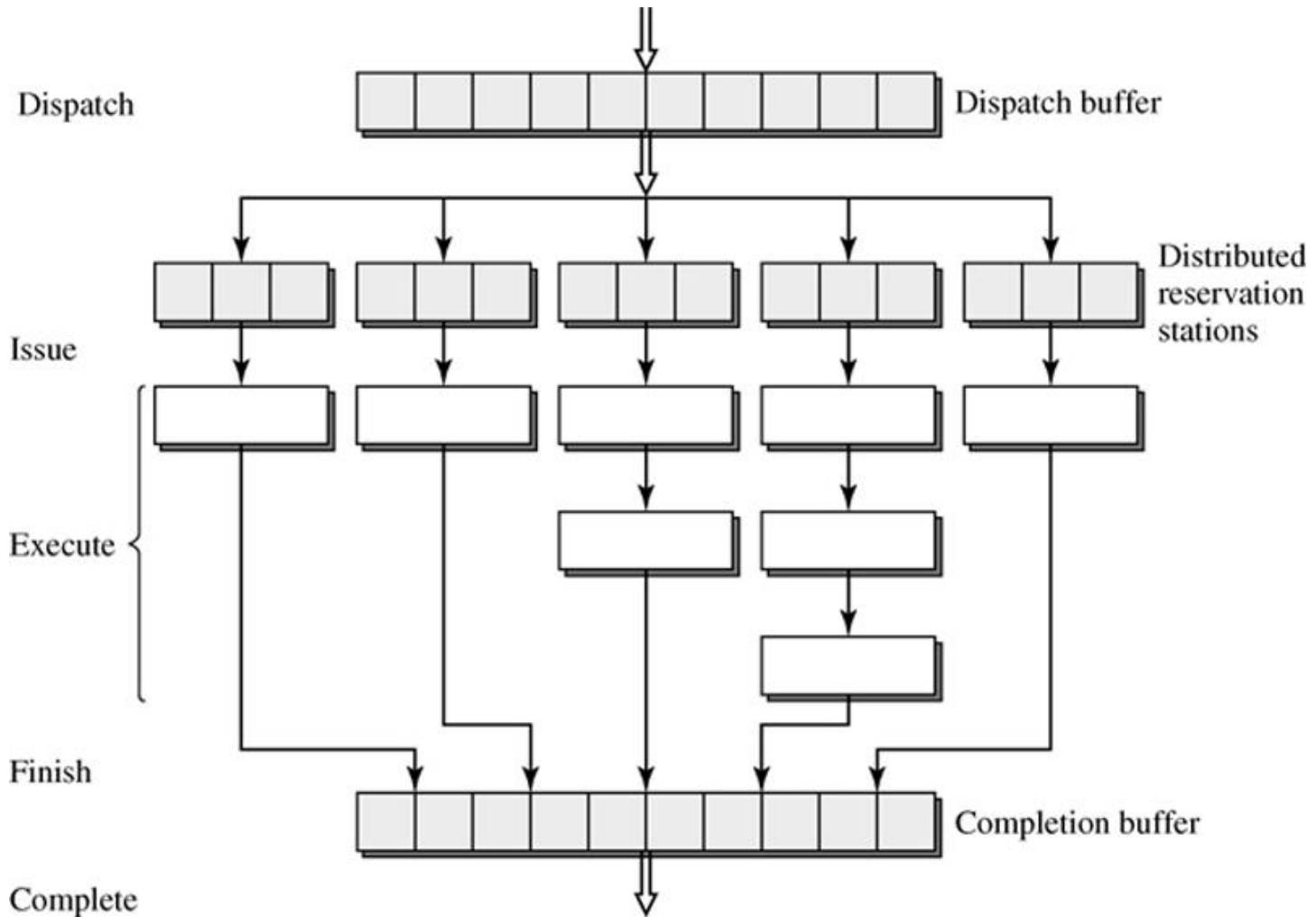
- **Reservation station:** προσωρινός καταχωρητής αποθήκευσης των αποκωδικοποιημένων εντολών που δεν έχουν διαθέσιμα όλα τα ορίσματά τους
 - Κεντρικός καταχωρητής (**centralized reservation station**)
 - » Για παράλληλες σωληνώσεις
 - Κατανεμημένοι καταχωρητές (**distributed reservation station**)
 - » Για ετερογενείς σωληνώσεις

Centralized reservation station

- Ενοποίηση των σταδίων dispatch και issue



Distributed reservation station



Εκτέλεση εντολών – Instruction Execution

- Σύγχρονες τάσεις:
 - Πολλές παράλληλες σωληνώσεις (δύσκολη η out-of-order εκτέλεση με bypassing εντολών)
 - Διαφοροποιημένες μεταξύ τους σωληνώσεις
 - Βαθιές σωληνώσεις
- Συνήθης καταμερισμός (δεν ακολουθεί τη στατιστική αναλογία των προγραμμάτων σε τύπους εντολών [40%-20%-40%]):
 - 4 μονάδες ALU
 - 1 μονάδα διακλάδωσης (μπορεί να εκτελέσει θεωρητικά (speculatively) > 1 εντολές διακλάδωσης)
 - 1 μονάδα ανάγνωσης/εγγραφής στη μνήμη (πολύ πολύπλοκη η υλοποίηση μνημών πολλαπλών εισόδων-εξόδων, μόνο με πολλαπλά banks)
 - Περισσότερες ειδικευμένες (και πιο αποδοτικές στην επίδοση) λειτουργικές μονάδες

Ολοκλήρωση και Αποδέσμευση εντολών – Instruction Completion and Retiring

- **Instruction Completion:** ενημέρωση της κατάστασης του μηχανήματος (machine state update)
- **Instruction Retiring:** ενημέρωση της μνήμης (memory state update)
- Αν η εντολή δεν περιλαμβάνει ενημέρωση της μνήμης, μετά το στάδιο ολοκλήρωσης, η εντολή αποδεσμεύεται
- Τα στάδια ολοκλήρωσης και αποδέσμευσης των εντολών πρέπει να εκτελέσουν τις εντολές εν σειρά
 - συνεπές arch. state
 - συνεπής κατάσταση μνήμης

Ολοκλήρωση και Αποδέσμευση εντολών – Instruction Completion and Retiring

- Τα στάδια εντολών:

- › Fetch
- › Decode
- › Dispatch
- › Issue
- › Execute
- › Finish
- › Complete
- › Retire

