Lecture-13 (ROB and Multi-threading) CS422-Spring 2018

Biswa@CSE-IITK

Cycle 62 (Scoreboard) vs 57 in Tomasulo

Instruction status:

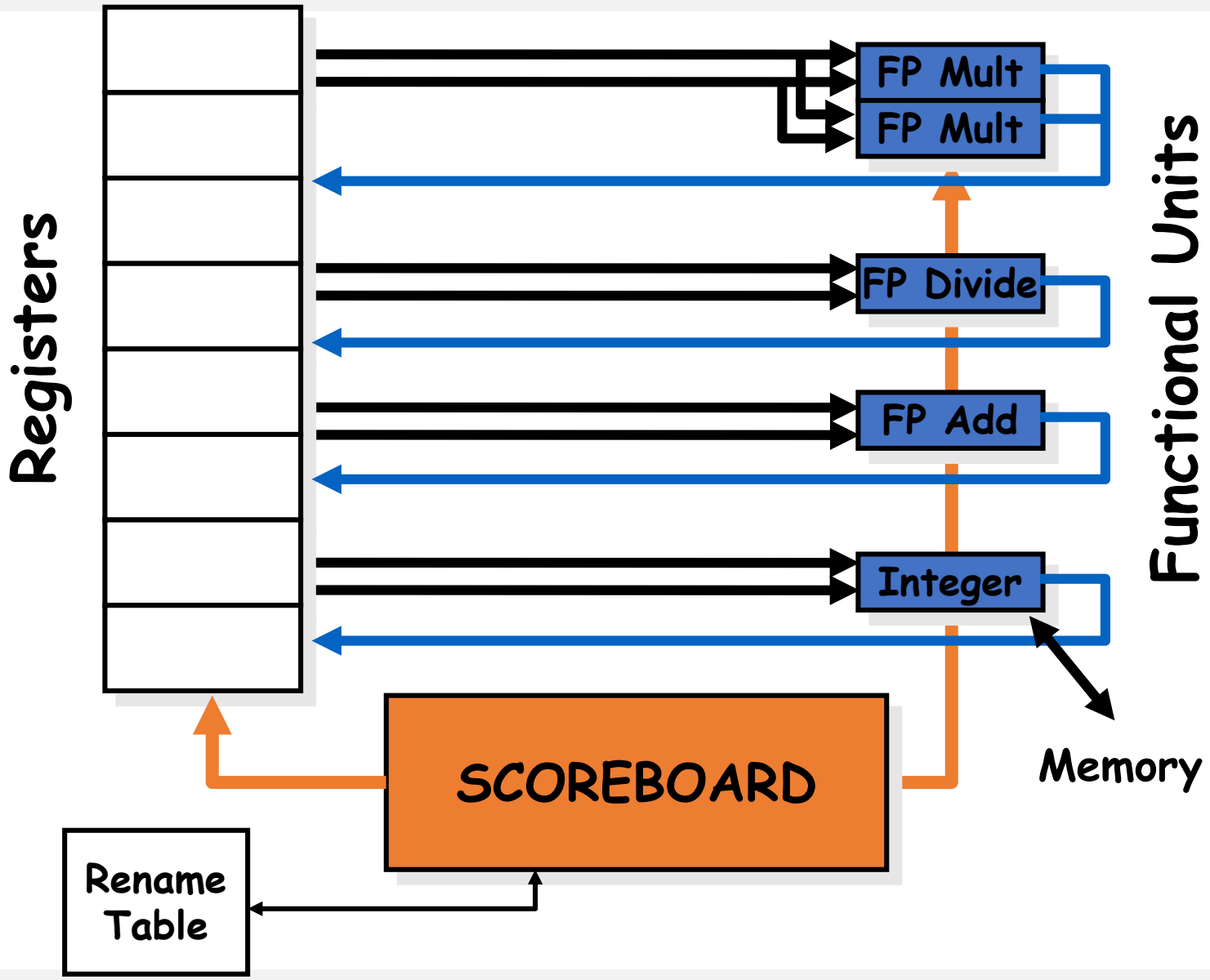
Instruction		*j*	*k*	*Issue*	*Read Oper*	*Exec Comp*	*Write Result*
LD	F6	34+	R2	1	2	3	4
LD	F2	45+	R3	5	6	7	8
MULTD	F0	F2	F4	6	9	19	20
SUBD	F8	F6	F2	7	9	11	12
DIVD	F10	F0	F6	8	21	61	62
ADDD	F6	F8	F2	13	14	16	22

Issue	*Exec Comp*	*Write Result*
1	3	4
2	4	5
3	15	16
4	7	8
5	56	57
6	10	11

Tomasulo vs Scoreboard

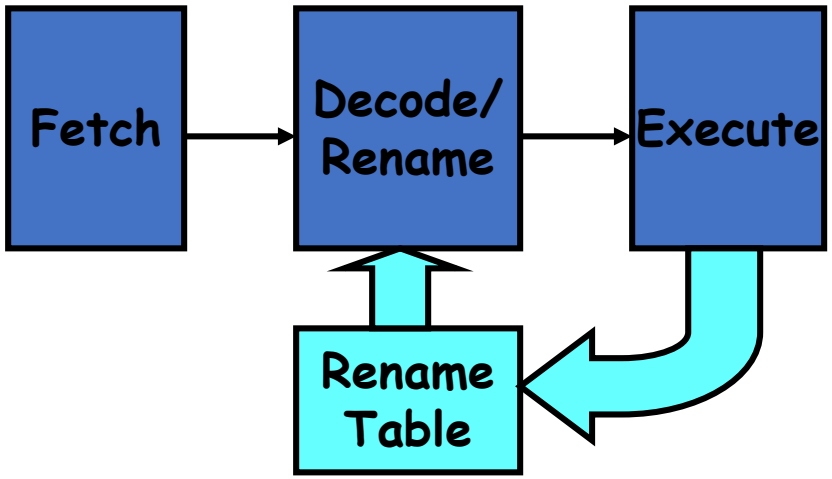
Pipelined Functional Units	Multiple Functional Units
(6 load, 3 store, 3 +, 2 x/÷)	(1 load/store, 1 + , 2 x, 1 ÷)
window size: ≤ 14 instructions	≤ 5 instructions
No issue on structural hazard	same
WAR: renaming avoids	stall completion
WAW: renaming avoids	stall issue
Broadcast results from FU	Write/read registers
Control: reservation stations	central scoreboard

Register Renaming + Scoreboard ?



Explicit Register Renaming

- Make use of a *physical* register file that is larger than number of registers specified by ISA
- Keep a translation table:
  - ISA register => physical register mapping
  - Physical register becomes free when not being used by any instructions in progress.



Explicit Register Renaming includes

- Rapid access to a table of translations
- A physical register file that has more registers than specified by the ISA
- Ability to figure out which physical registers are free.
  - No free registers $\Rightarrow$ stall on issue
- Many modern architectures use explicit register renaming + Tomasulo-like reservation stations to control execution.
- Relationship between #registers and #RS entries: Piazza +1

What About Exceptions and Interrupts?

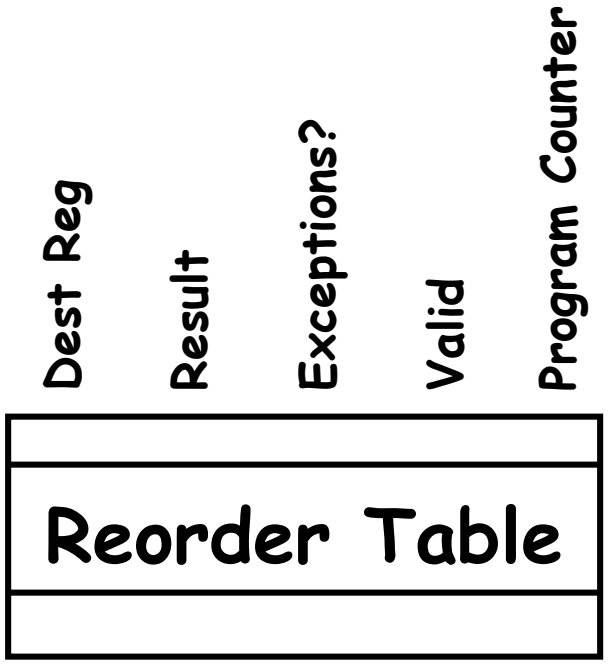
- Both Scoreboard and Tomasulo have:
  - In-order issue, out-of-order execution, out-of-order completion
- Recall: An interrupt or exception is precise if there is a single instruction for which:
  - All instructions before that have committed their state
  - No following instructions (including the interrupting instruction) have modified any state.

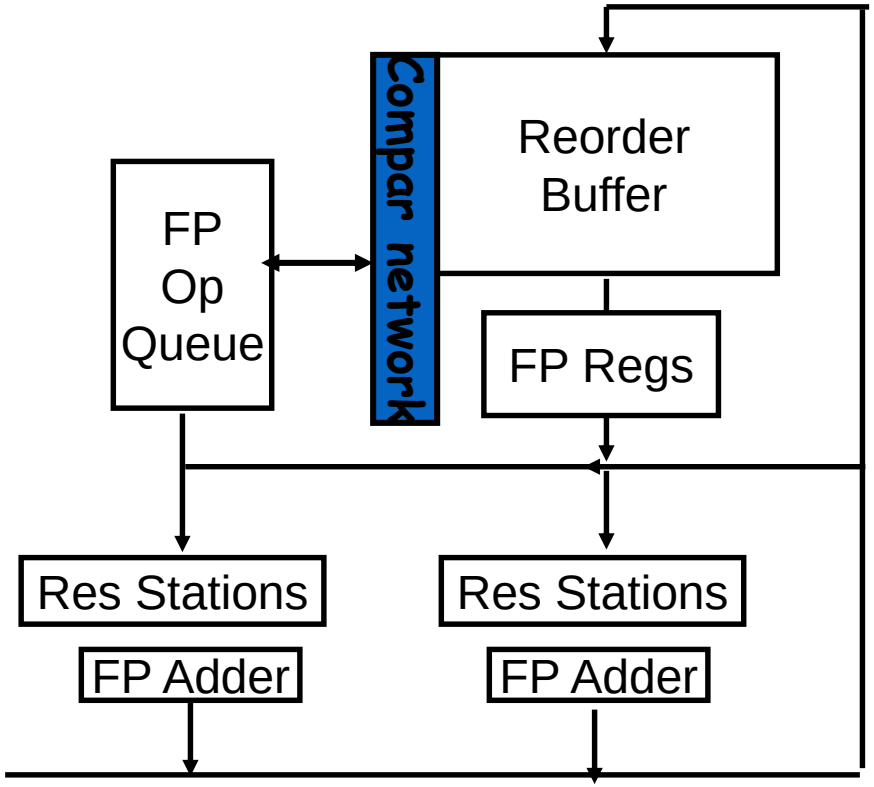
- Need way to resynchronize execution with instruction stream (I.e. with issue-order)

Re-order Buffer

- Out-of-order commit: What about precise exceptions ?
- It would be great to have in-order commit with O3 execute.
- The process of an instruction leaving the ROB will now be called as commit.
- To preserve precise exceptions, a result is written into the register file only when the instruction commits.
- Until then, the result is saved in a temporary register in the ROB.

Re-order Buffer (ROB)





Speculative Tomasulo with ROB

1. Issue—get instruction from FP Op Queue

   If reservation station and reorder buffer slot free, issue instr & send operands & reorder buffer 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 in reservation station, execute; checks RAW (sometimes called "issue")

3. Write result—finish execution (WB)

   Write on Common Data Bus to all awaiting FUs
   & reorder buffer; mark reservation station available.

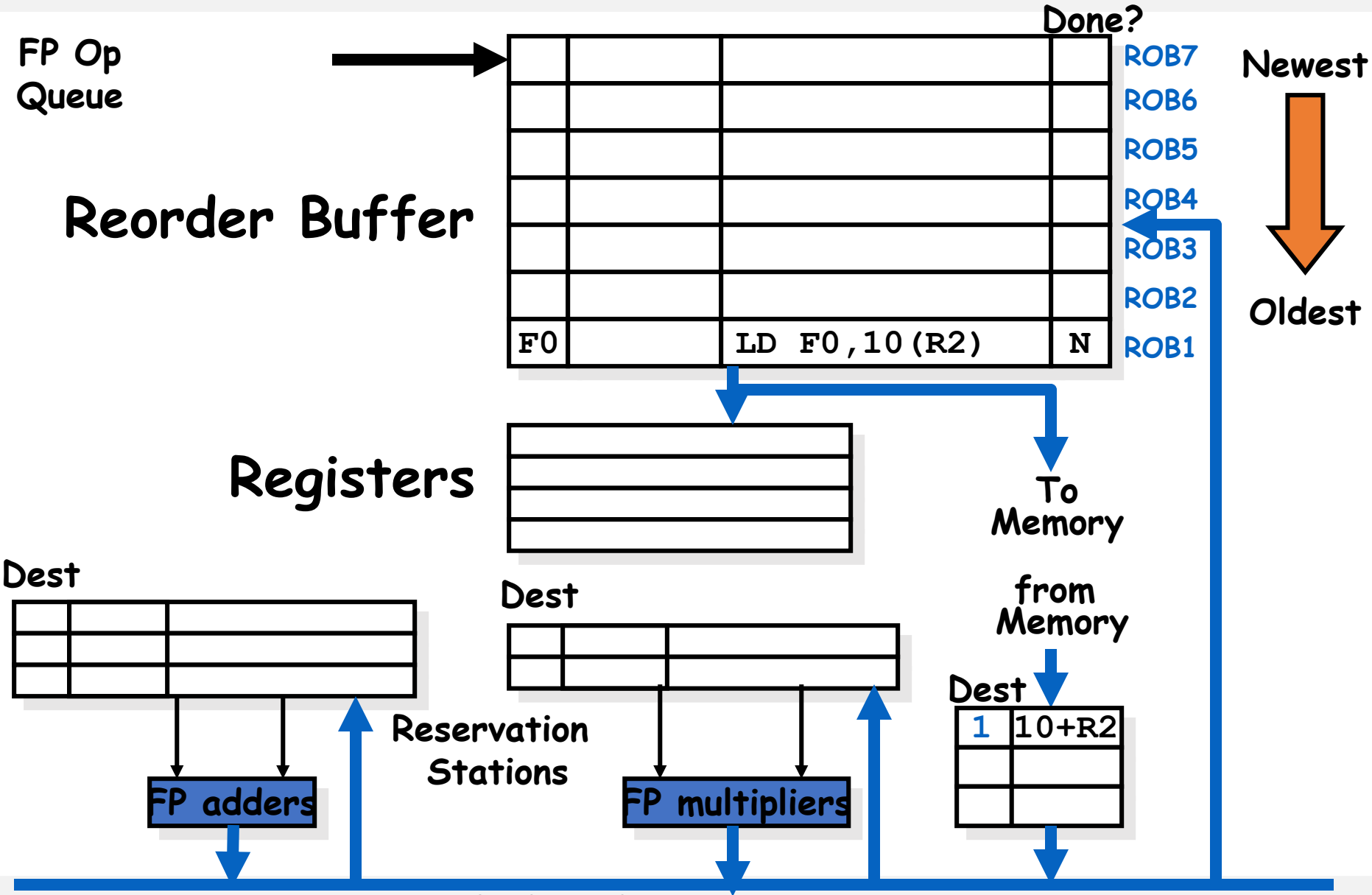
4. Commit—update register with reorder result

   When instr. at head of reorder buffer & result present, update register with result (or store to memory) and remove instr from reorder buffer. Mispredicted branch flushes reorder buffer (sometimes called "graduation")

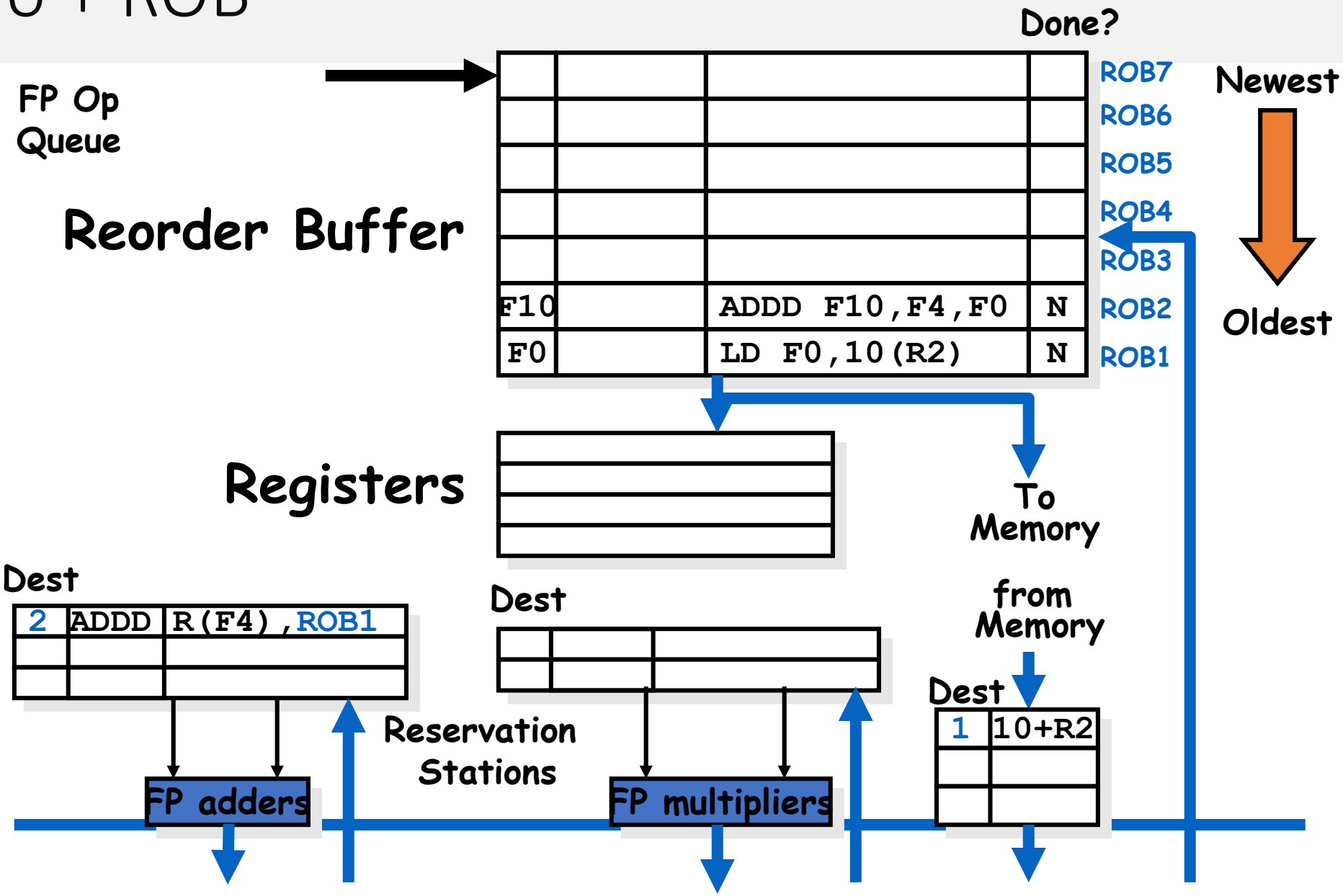
ROB Entry

- Type of Instruction
- Destination: None, Memory Address, Register including ROB entry
- Value and its presence/absence.
- Reservation station tags and true register tags are now ids of entries in the ROB.

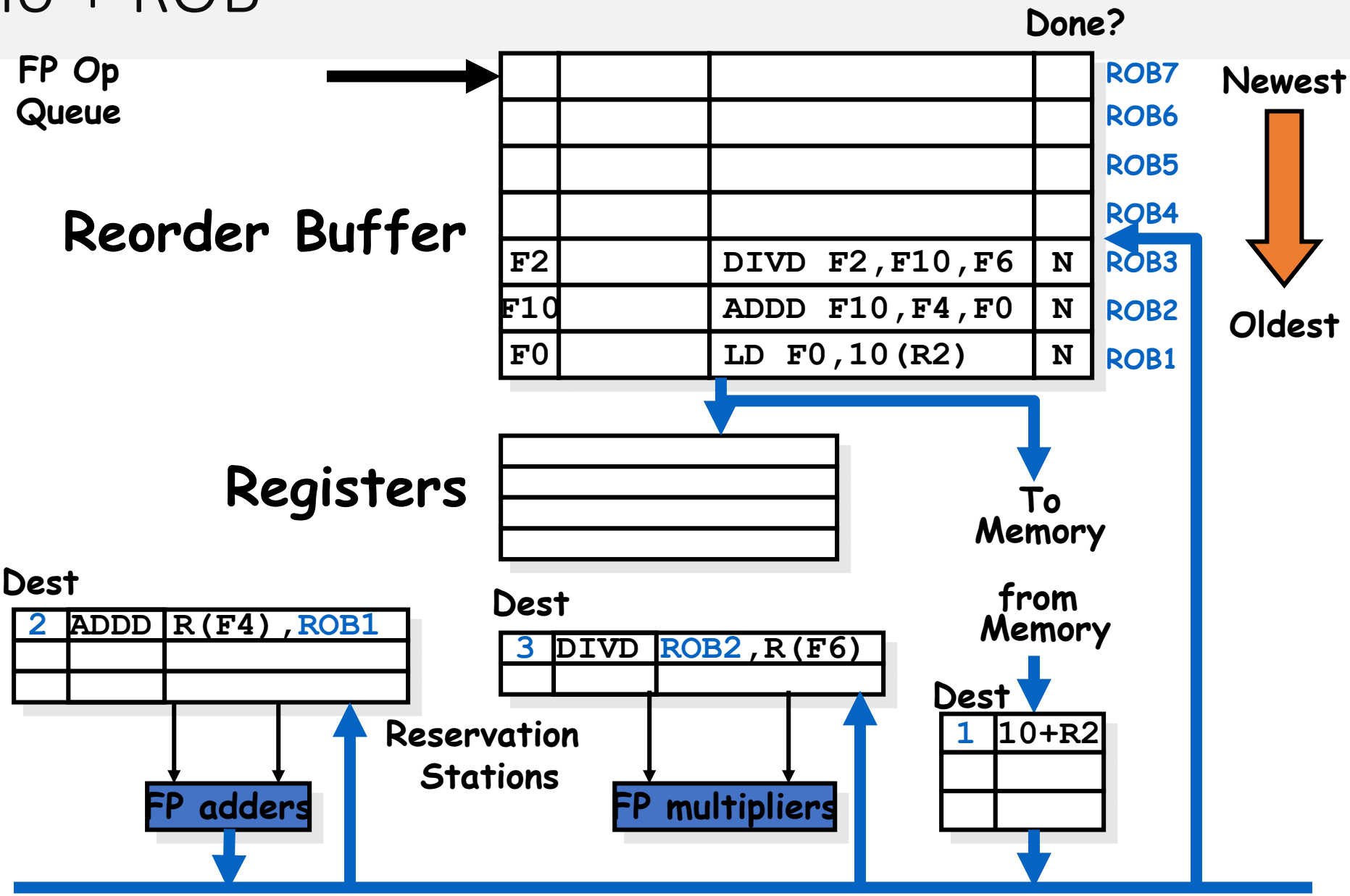
Tomasulo + ROB



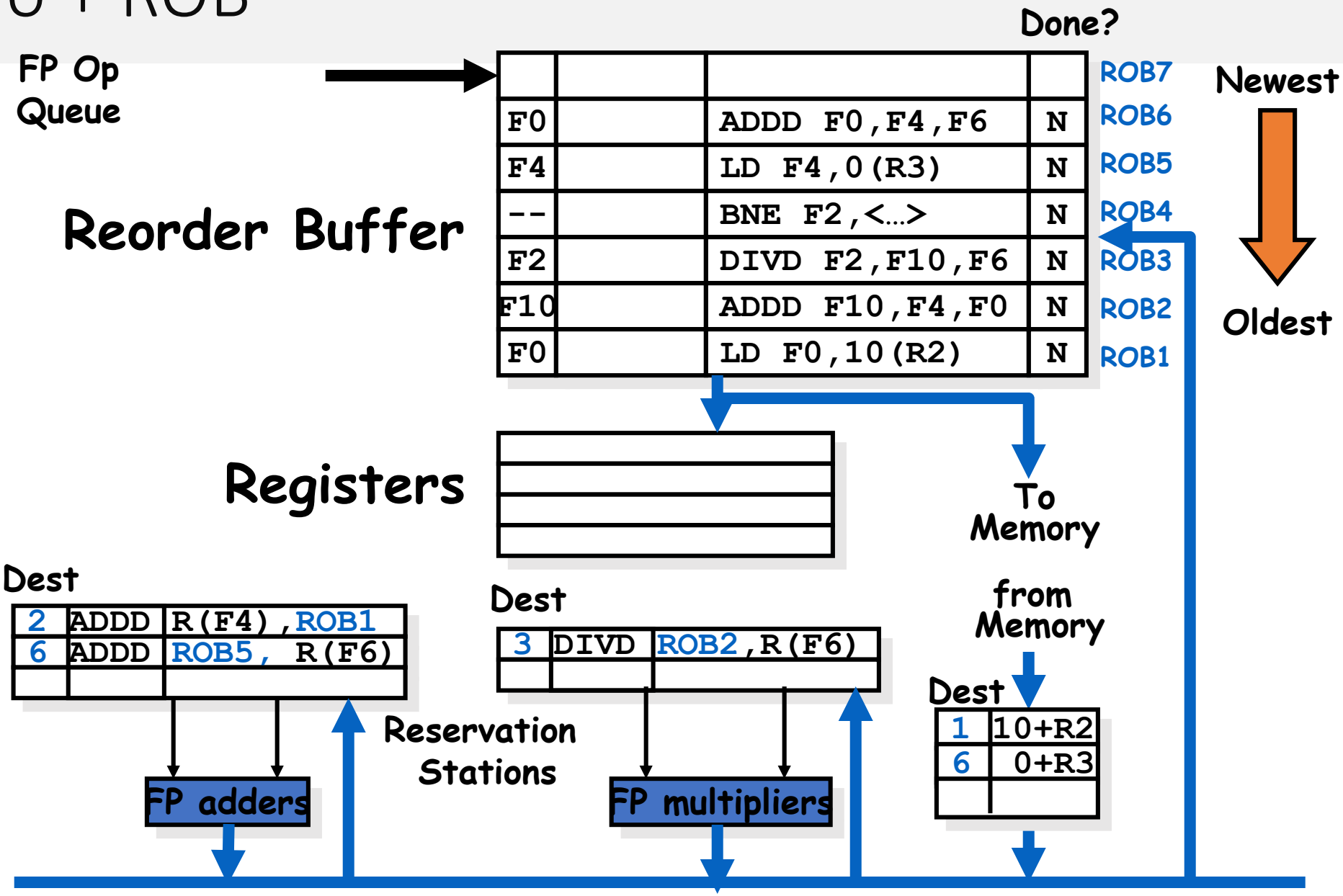
Tomasulo + ROB



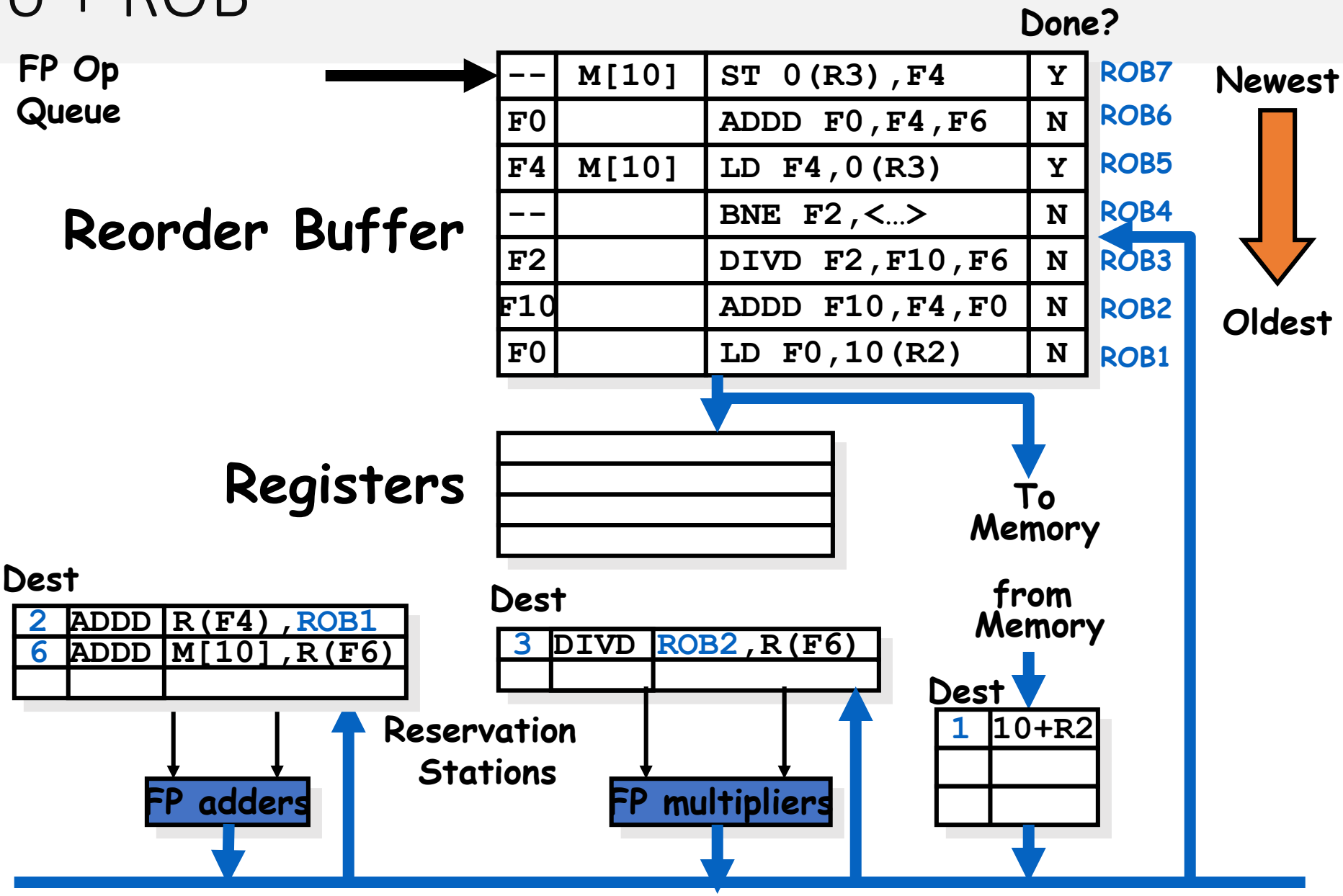
Tomasulo + ROB



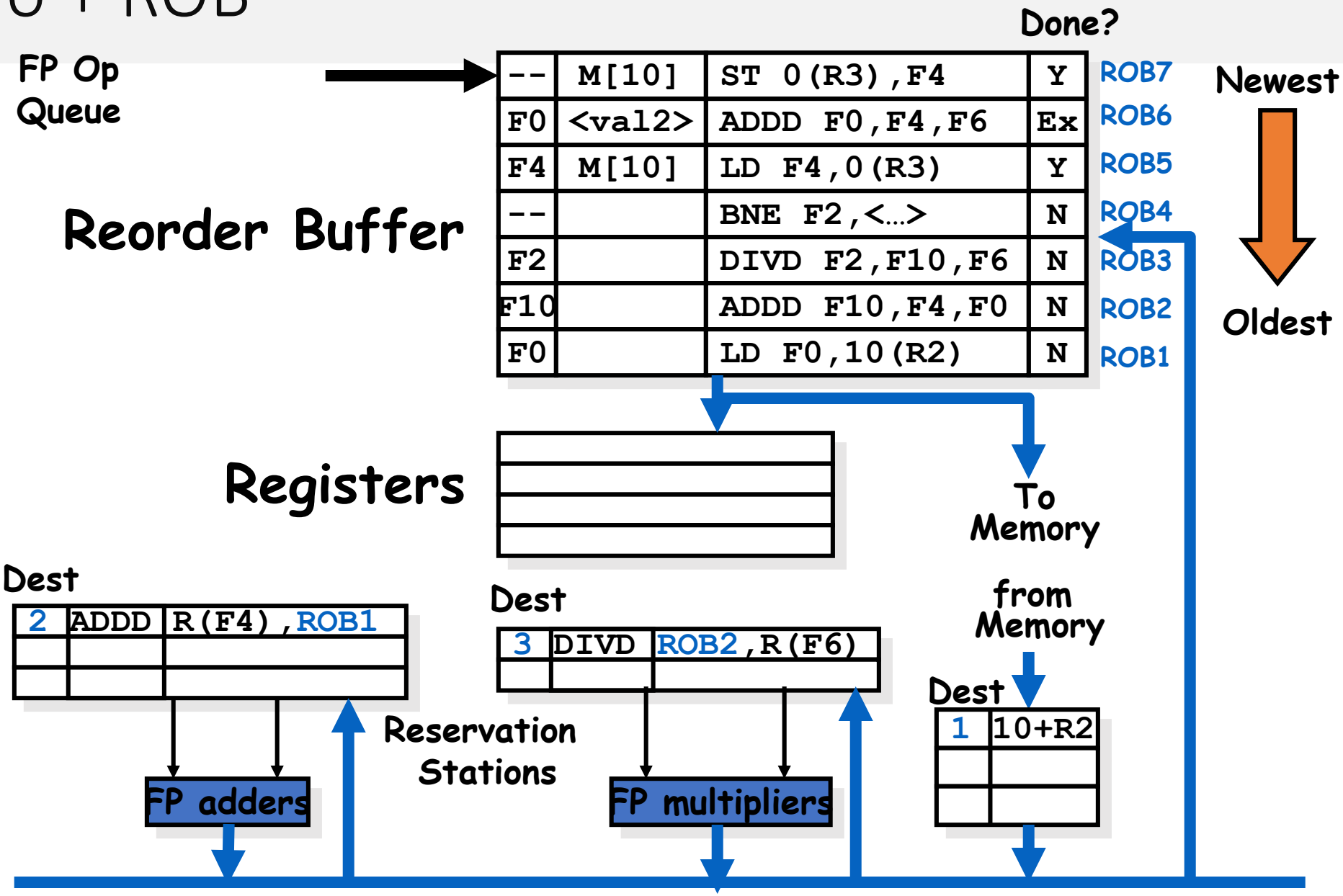
Tomasulo + ROB



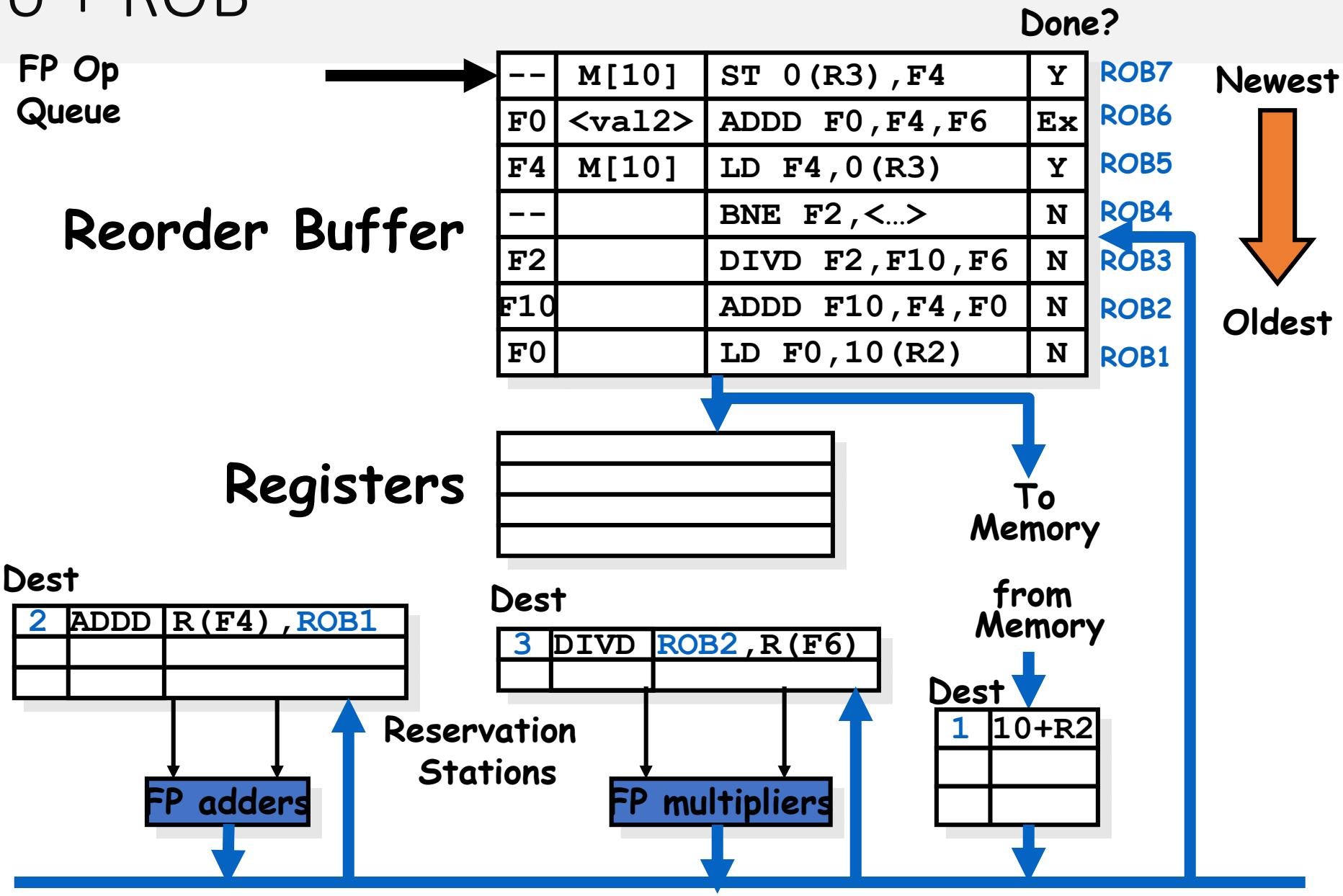
Tomasulo + ROB



Tomasulo + ROB



Tomasulo + ROB



Limits of ILP

Limits to ILP

Initial HW Model here; MIPS compilers.

Assumptions for ideal/perfect machine to start:

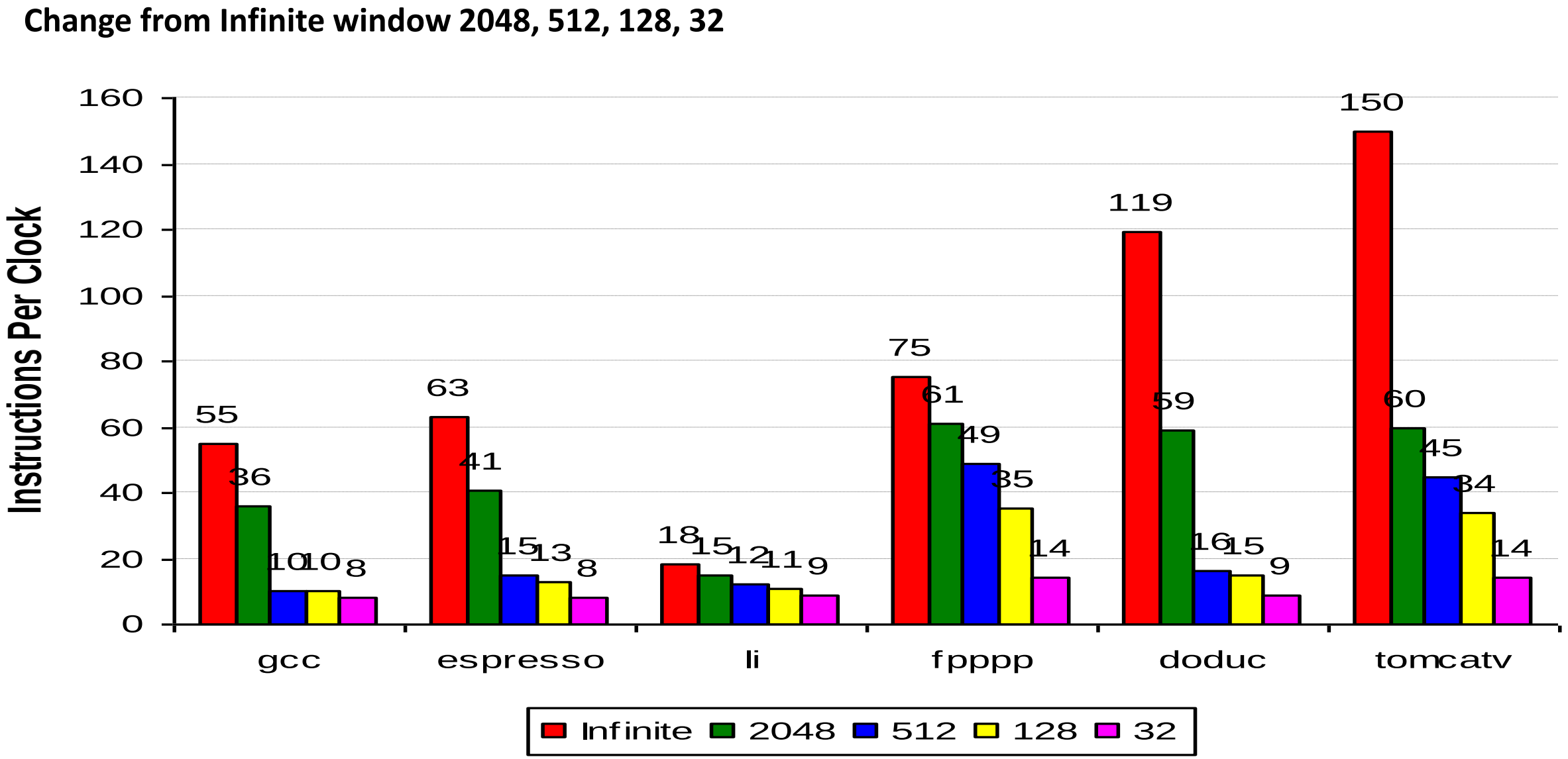
1. *Register renaming* – infinite virtual registers
   ⇒ all register WAW & WAR hazards are avoided

2. *Branch prediction* – perfect; no mispredictions

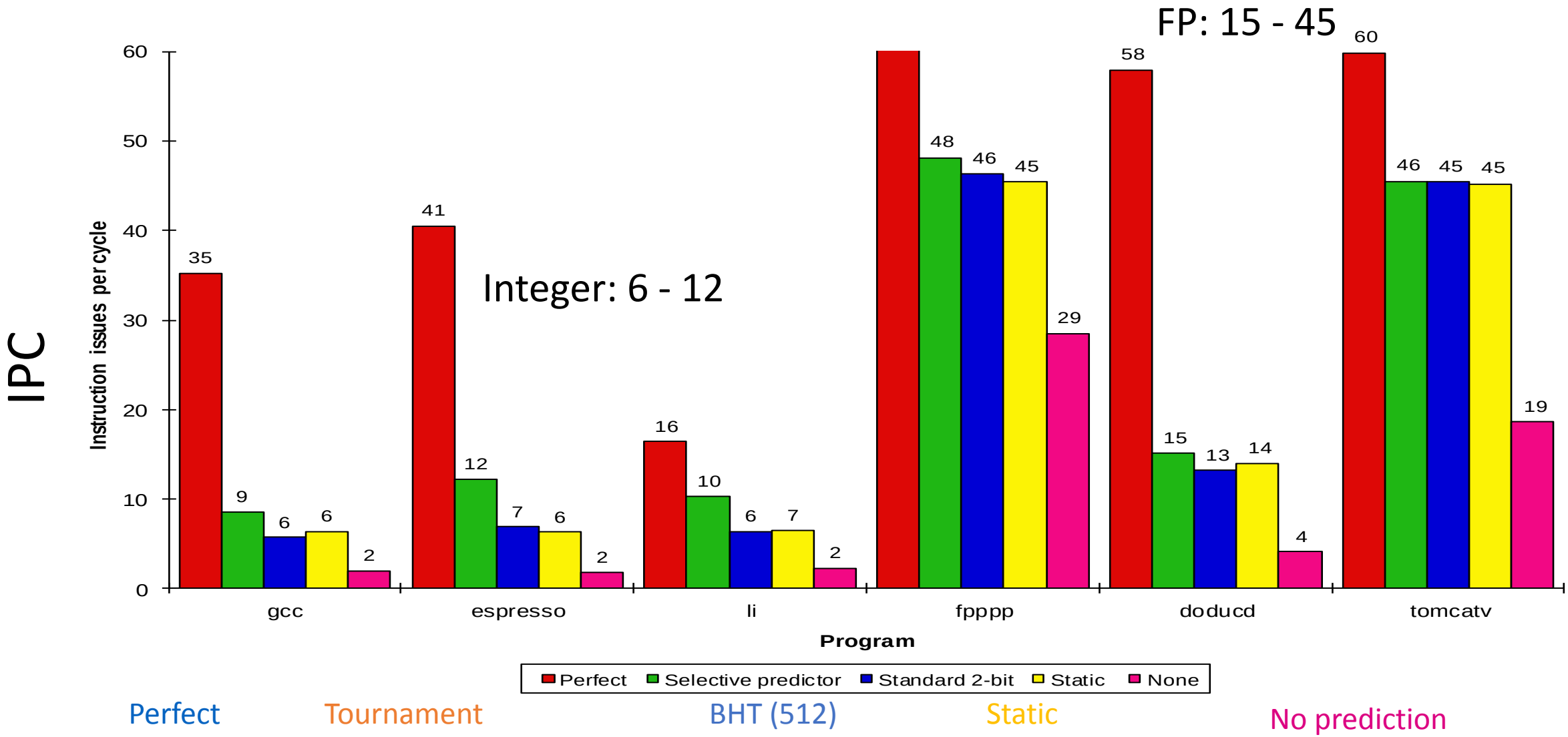
3. *Jump prediction* – all jumps perfectly predicted (returns, case statements)
   2 & 3 ⇒ no control dependencies; perfect speculation & an unbounded buffer of instructions available

Also: perfect caches; 1 cycle latency for all instructions (FP *,/); unlimited instructions issued/clock cycle;

Window Impact on IPC



Branch Impact



Perfect Tournament BHT (512) Static No prediction

Beyond ILP

- There can be much higher natural parallelism in some applications
  (e.g., Database or Scientific codes)
- Explicit Thread Level Parallelism or Data Level Parallelism
- Thread: instruction stream with own PC and data
  - thread may be a process part of a parallel program of multiple processes, or it may be an independent program
  - Each thread has all the state (instructions, data, PC, register state, and so on) necessary to allow it to execute
- Data Level Parallelism: Perform identical operations on data, and lots of data

TLP – Thread Level Parallelism

- ILP exploits implicit parallel operations within a loop or straight-line code segment

- TLP explicitly represented by the use of multiple threads of execution that are inherently parallel

- Goal: Use multiple instruction streams to improve
  1. Throughput of computers that run many programs
  2. Execution time of multi-threaded programs

- TLP could be more cost-effective to exploit than ILP

Multiple Threads in Execution

- Multithreading: multiple threads to share the functional units of 1 processor via overlapping
  - processor must duplicate independent state of each thread e.g., a separate copy of register file, a separate PC, and for running independent programs, a separate page table
  - memory shared through the virtual memory mechanisms, which already support multiple processes
  - HW for fast thread switch; much faster than full process switch ≈ 100s to 1000s of clocks

- When switch?
  - Alternate instruction per thread (fine grain)
  - When a thread is stalled, perhaps for a cache miss, another thread can be executed (coarse grain)

Fine-grained

- Switches between threads on each instruction, causing the execution of multiples threads to be interleaved
- Usually done in a round-robin fashion, skipping any stalled threads
- CPU must be able to switch threads every clock
- Advantage is it can hide both short and long stalls, since instructions from other threads executed when one thread stalls
- Disadvantage is it slows down execution of individual threads, since a thread ready to execute without stalls will be delayed by instructions from other threads
- Used on Sun's Niagara

Coarse-grained

- Switches threads only on costly stalls, such as L2 cache misses

- Advantages
  - Relieves need to have very fast thread-switching
  - Doesn't slow down thread, since instructions from other threads issued only when the thread encounters a costly stall

- Disadvantage is hard to overcome throughput losses from shorter stalls, due to pipeline start-up costs
  - Since CPU issues instructions from 1 thread, when a stall occurs, the pipeline must be emptied or frozen
  - New thread must fill pipeline before instructions can complete

- Because of this start-up overhead, coarse-grained multithreading is better for reducing penalty of high cost stalls, where pipeline refill << stall time

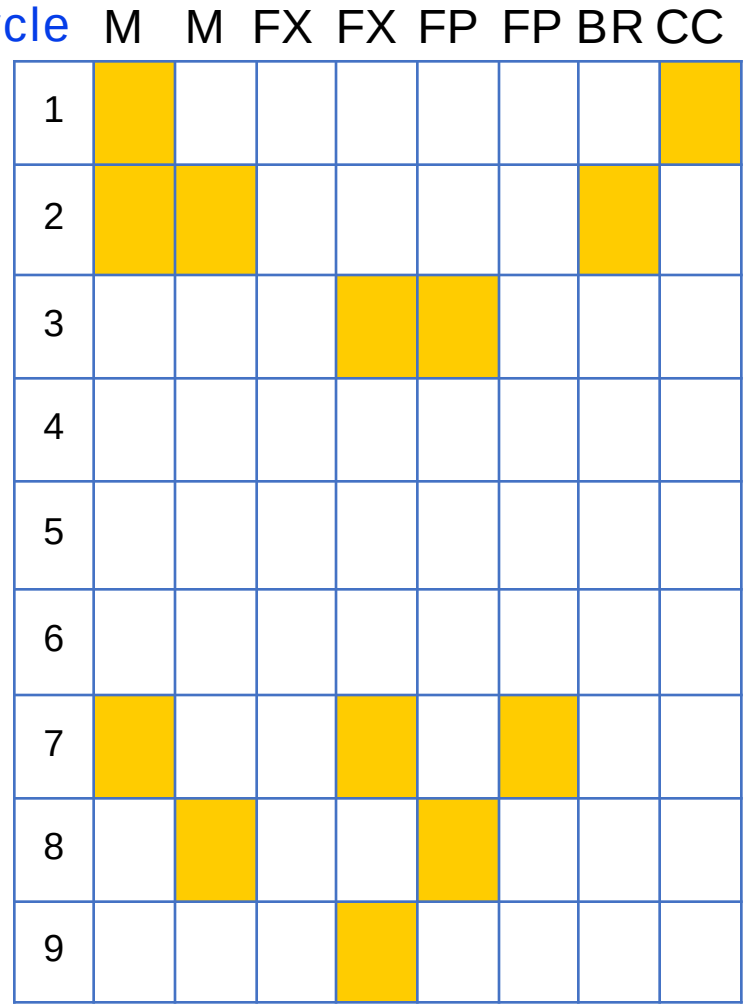
- Used in IBM AS/400, Alewife

TLP + ILP

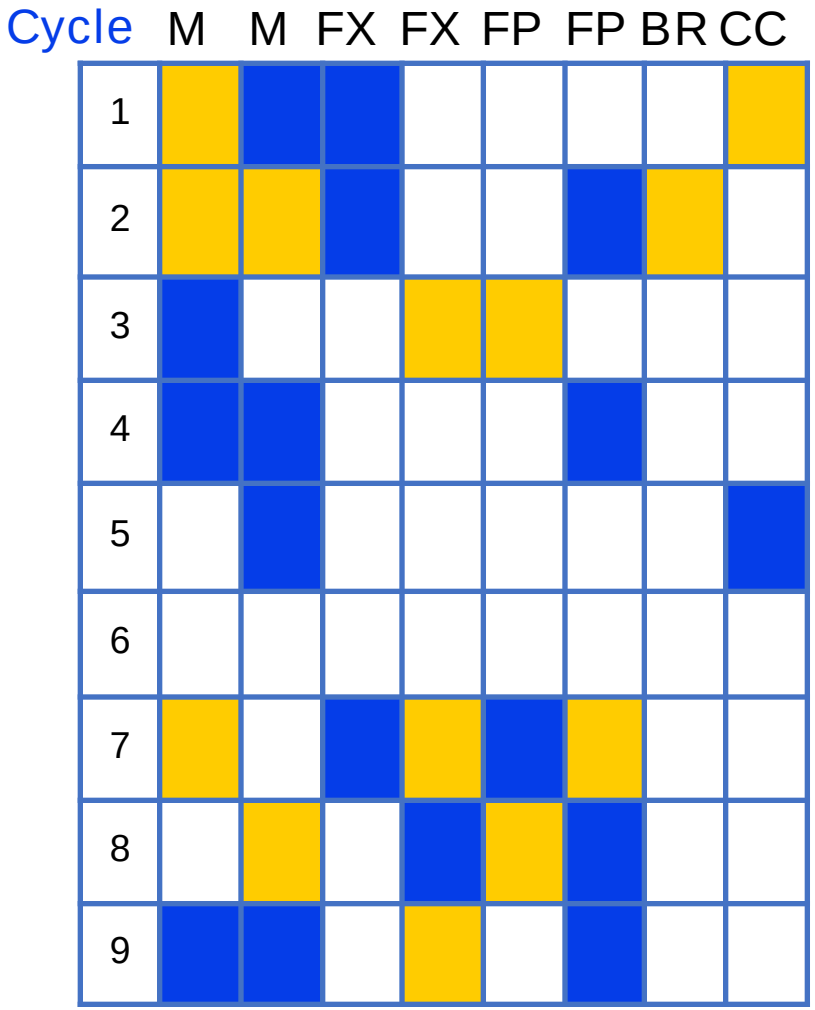
- TLP and ILP exploit two different kinds of parallel structure in a program
- Could a processor oriented at ILP to exploit TLP?
  - functional units are often idle in data path designed for ILP because of either stalls or dependences in the code
- Could the TLP be used as a source of independent instructions that might keep the processor busy during stalls?
- Could TLP be used to employ the functional units that would otherwise lie idle when insufficient ILP exists?

SMT

One thread, 8 units



Two threads, 8 units



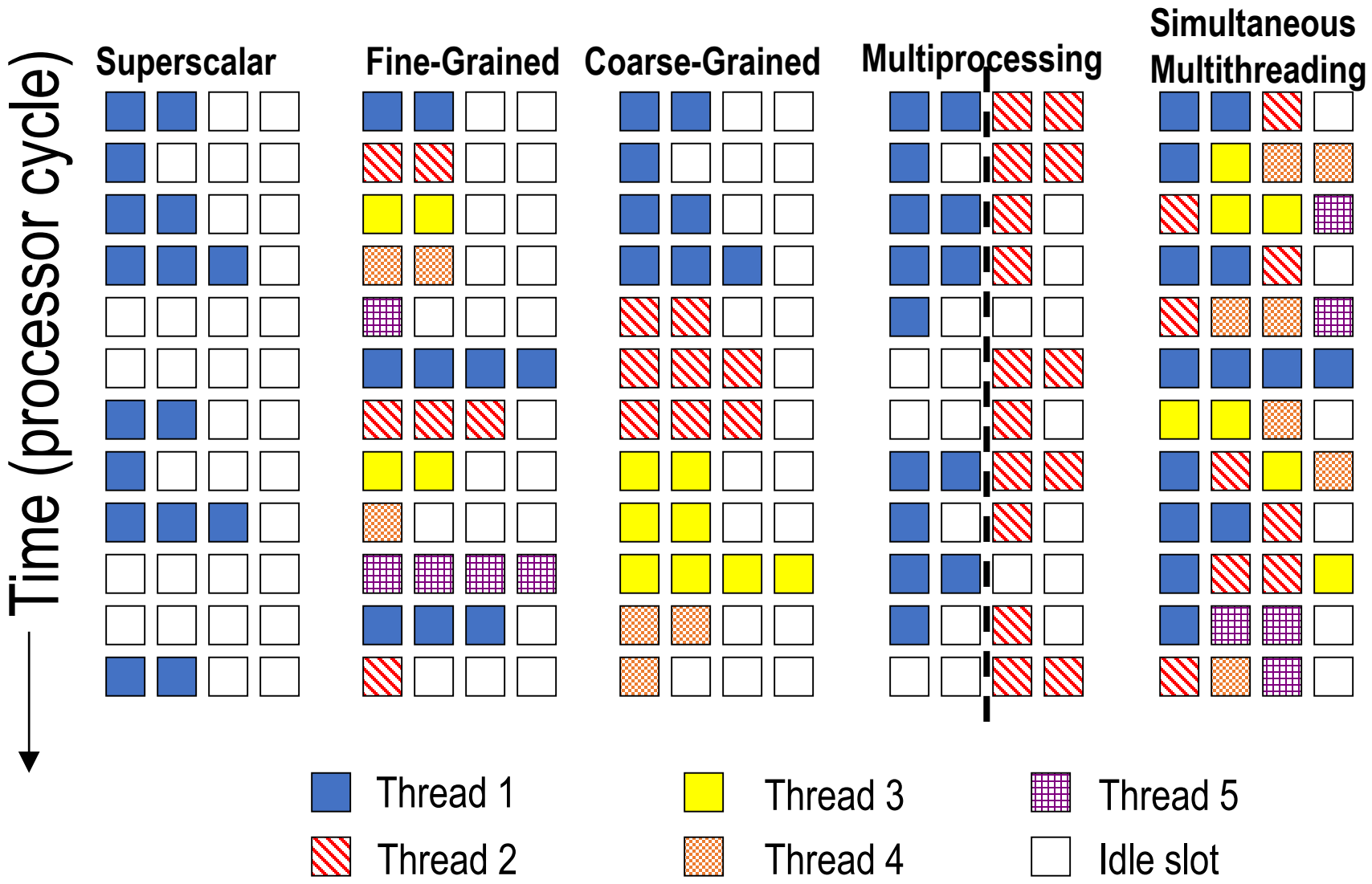
M = Load/Store, FX = Fixed Point, FP = Floating Point, BR = Branch, CC = Condition Codes

SMT

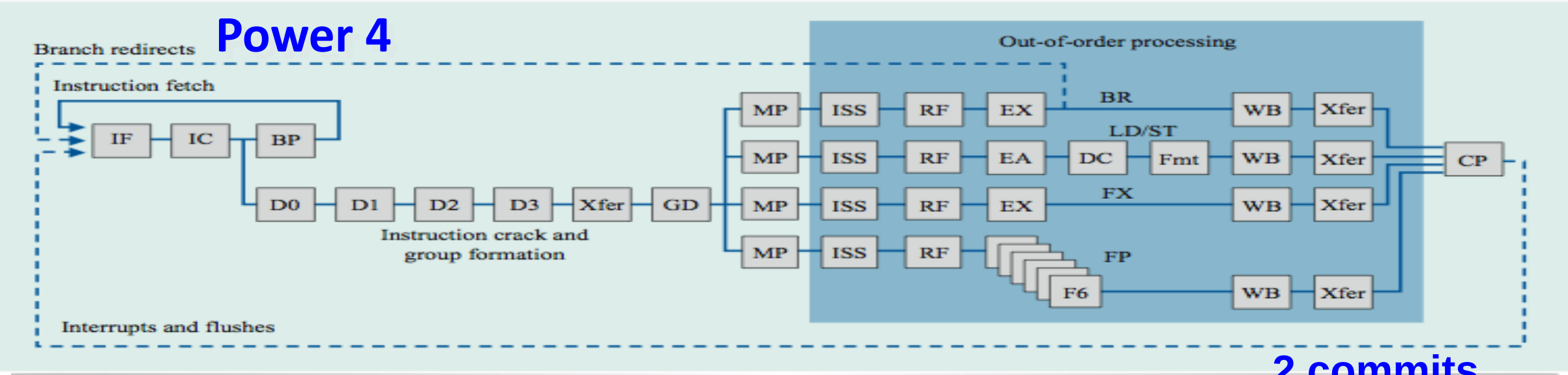
- Simultaneous multithreading (SMT): insight that dynamically scheduled processor already has many HW mechanisms to support multithreading
  - Large set of virtual registers that can be used to hold the register sets of independent threads
  - Register renaming provides unique register identifiers, so instructions from multiple threads can be mixed in data path without confusing sources and destinations across threads
  - Out-of-order completion allows the threads to execute out of order, and get better utilization of the HW

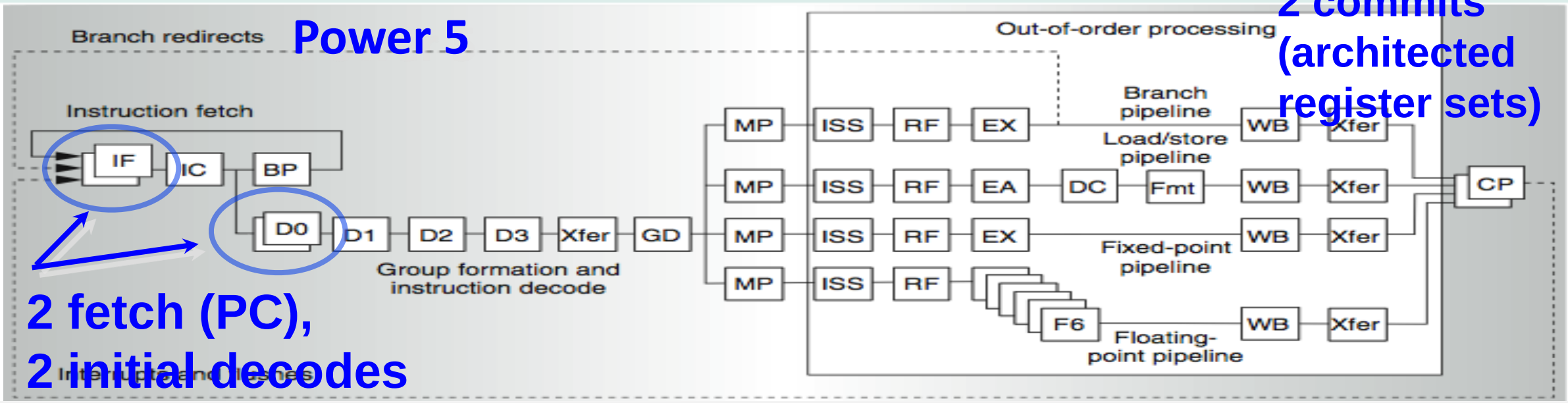
- Just adding a per thread renaming table and keeping separate PCs
  - Independent commitment can be supported by logically keeping a separate reorder buffer for each thread

All in One


Superscalar
Fine-Grained
Coarse-Grained
Multiprocessing
Simultaneous Multithreading
Time (processor cycle)
Thread 1
Thread 2
Thread 3
Thread 4
Thread 5
Idle slot

IBM POWER 4 and 5





Power 5 Data flow