

---

# Handout 2 – ILP: Part B

# Review from Last Time #1

---

- **Leverage Implicit Parallelism for Performance: Instruction Level Parallelism**
- **Loop unrolling by compiler to increase ILP**
- **Branch prediction to increase ILP**
- **Dynamic HW exploiting ILP**
  - Works when can't know dependence at compile time
  - Can hide L1 cache misses
  - Code for one machine runs well on another

# Review from Last Time #2

---

- **Reservations stations: *renaming* to larger set of registers + buffering source operands**
  - Prevents registers as bottleneck
  - Avoids WAR, WAW hazards
  - Allows loop unrolling in HW
- **Not limited to basic blocks (integer units gets ahead, beyond branches)**
- **Helps cache misses as well**
- **Lasting Contributions**
  - Dynamic scheduling
  - Register renaming
  - Load/store disambiguation
- **360/91 descendants are Pentium 4, Power 5, AMD Athlon/Opteron, ...**

# Outline

---

- **ILP**
- **Speculation**
- **Speculative Tomasulo Example**
- **Exceptions**
- **Issue Window vs. Reorder Buffer**
- **Simultaneous Multi-threading**

# Speculation to greater ILP

---

- **Greater ILP: Overcome control dependence by hardware speculating on outcome of branches and executing program as if guesses were correct**
  - **Speculation**  $\Rightarrow$  **fetch, issue, and execute instructions** as if branch predictions were always correct
  - **Dynamic scheduling**  $\Rightarrow$  deal with instruction scheduling
- **Essentially a data flow execution model:**
  - Operations execute as soon as their operands are available

# Speculation to greater ILP

---

- **3 components of HW-based speculation:**
  - 1. Dynamic branch prediction to choose which instructions to execute**
  - 2. Speculation to allow execution of instructions before control dependences are resolved**
    - + ability to undo effects of incorrectly speculated sequence
  - 3. Dynamic scheduling to deal with scheduling of different combinations of basic blocks**

# Adding Speculation to Tomasulo

---

- Must separate execution from allowing instruction to finish or “commit”
- This additional step called **instruction commit**
- When an instruction is no longer speculative, allow it to update the register file or memory
- Requires additional set of buffers to hold results of instructions that have finished execution but have not committed
- This **reorder buffer (ROB)** is also used to pass results among instructions that may be speculated

# Reorder Buffer (ROB)

---

- In Tomasulo's algorithm, once an instruction writes its result, any subsequently issued instructions will find result in the register file
- With speculation, the register file is not updated until the instruction commits
  - (we know definitively that the instruction should execute)
- Thus, the ROB supplies operands in interval between completion of instruction execution and instruction commit
  - ROB is a source of operands for instructions, just as reservation stations (RS) provide operands in Tomasulo's algorithm
  - ROB extends architected registers like RS



# Reorder Buffer Entry

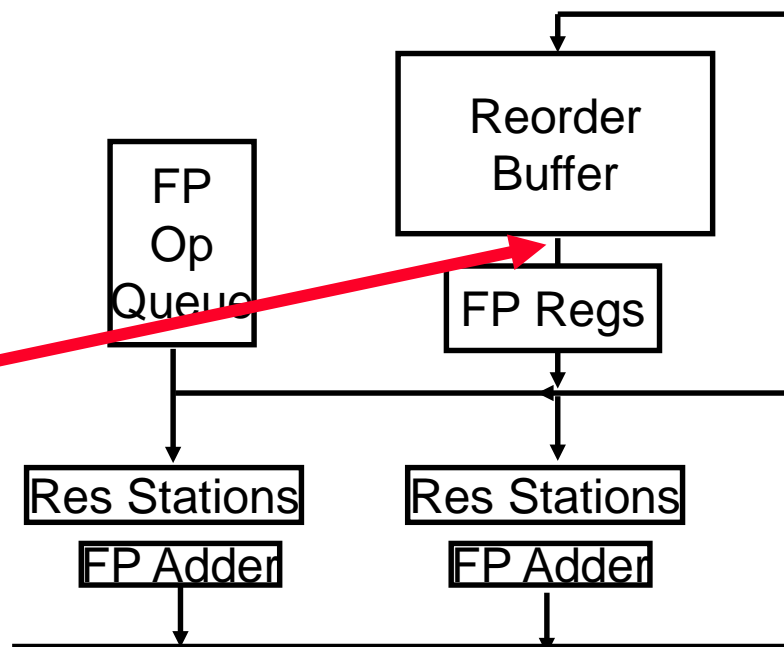
---

- **Each entry in the ROB contains four fields:**
  - 1. Instruction type**
    - a branch (has no destination result), a store (has a memory address destination), or a register operation (ALU operation or load, which has register destinations)
  - 2. Destination**
    - Register number (for loads and ALU operations) or memory address (for stores) where the instruction result should be written
  - 3. Value**
    - Value of instruction result until the instruction commits
  - 4. Ready**
    - Indicates that instruction has completed execution, and the value is ready

# Reorder Buffer operation

- Holds instructions in FIFO order, exactly as issued
- When instructions complete, results placed into ROB
  - Supplies operands to other instruction between execution complete & commit  $\Rightarrow$  more registers like RS
  - Tag results with ROB buffer number instead of reservation station
- Instructions **commit**  $\Rightarrow$  values at head of ROB placed in registers
- As a result, easy to undo speculated instructions on mispredicted branches or on exceptions

Commit path



# Recall: 4 Steps of Speculative Tomasulo Algorithm

---

## 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”)

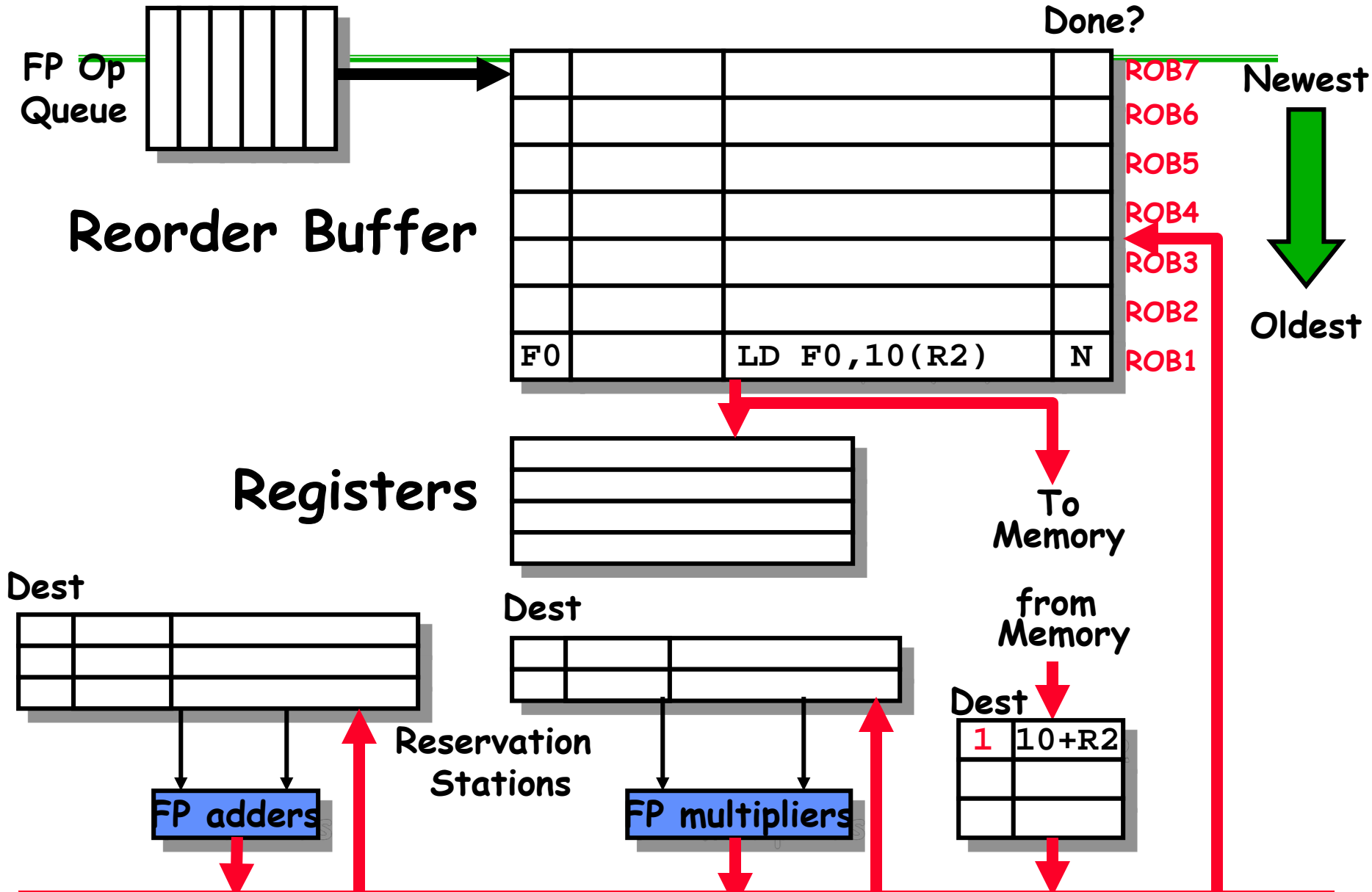
## 3. Write result—finish execution (WB)

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

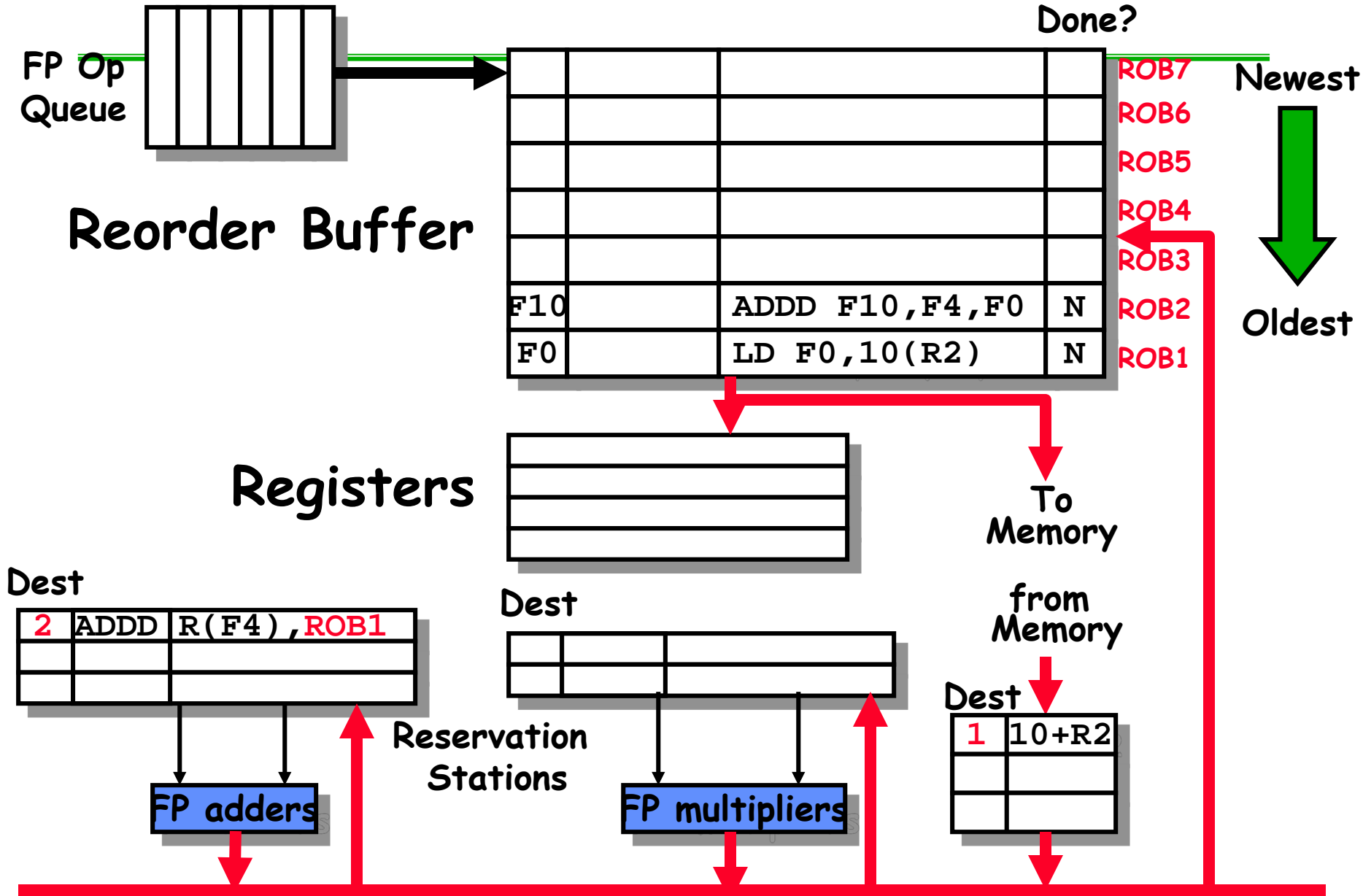
## 4. Commit—update register with reorder result

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”)

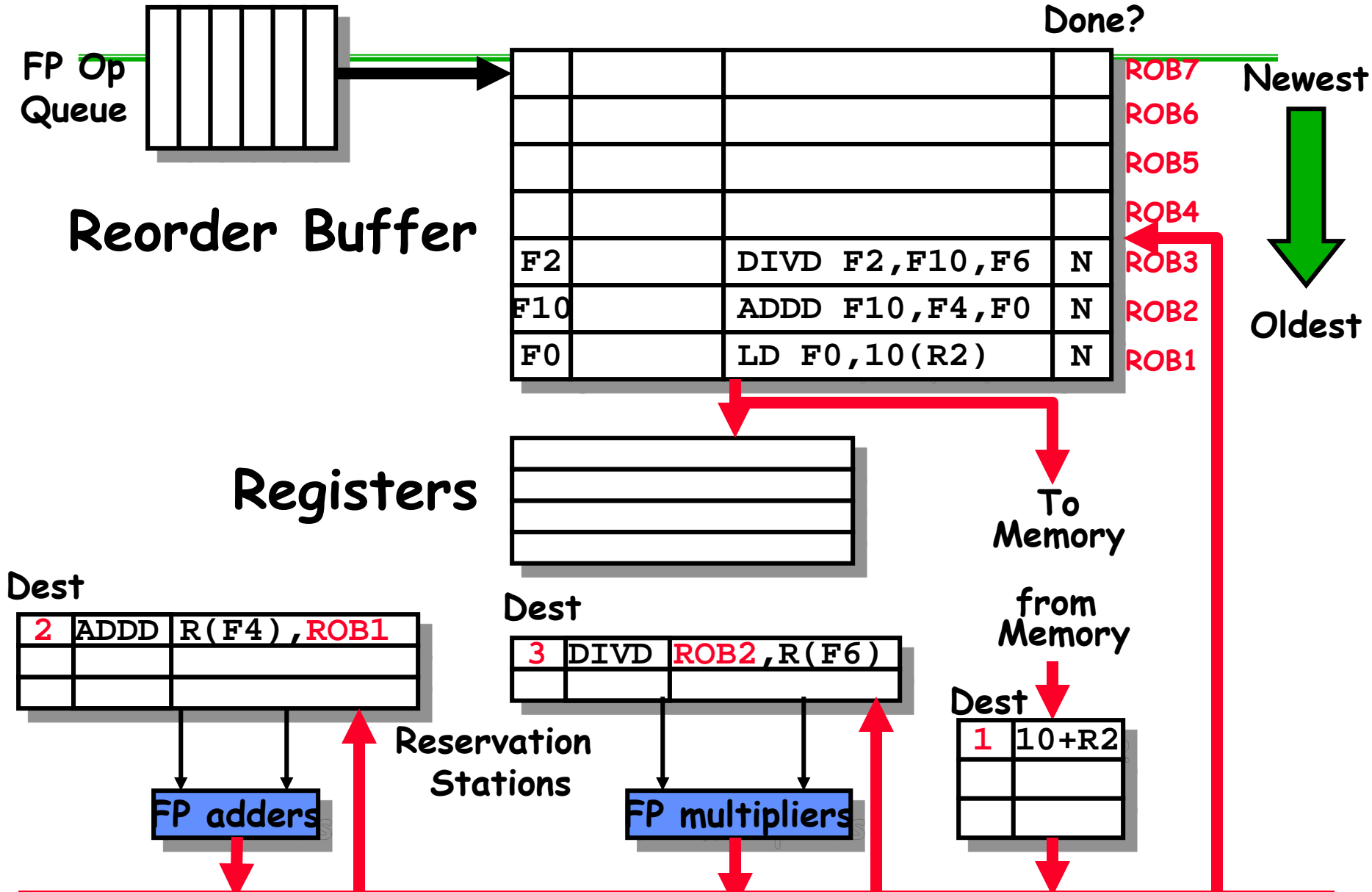
# Tomasulo With Reorder buffer:



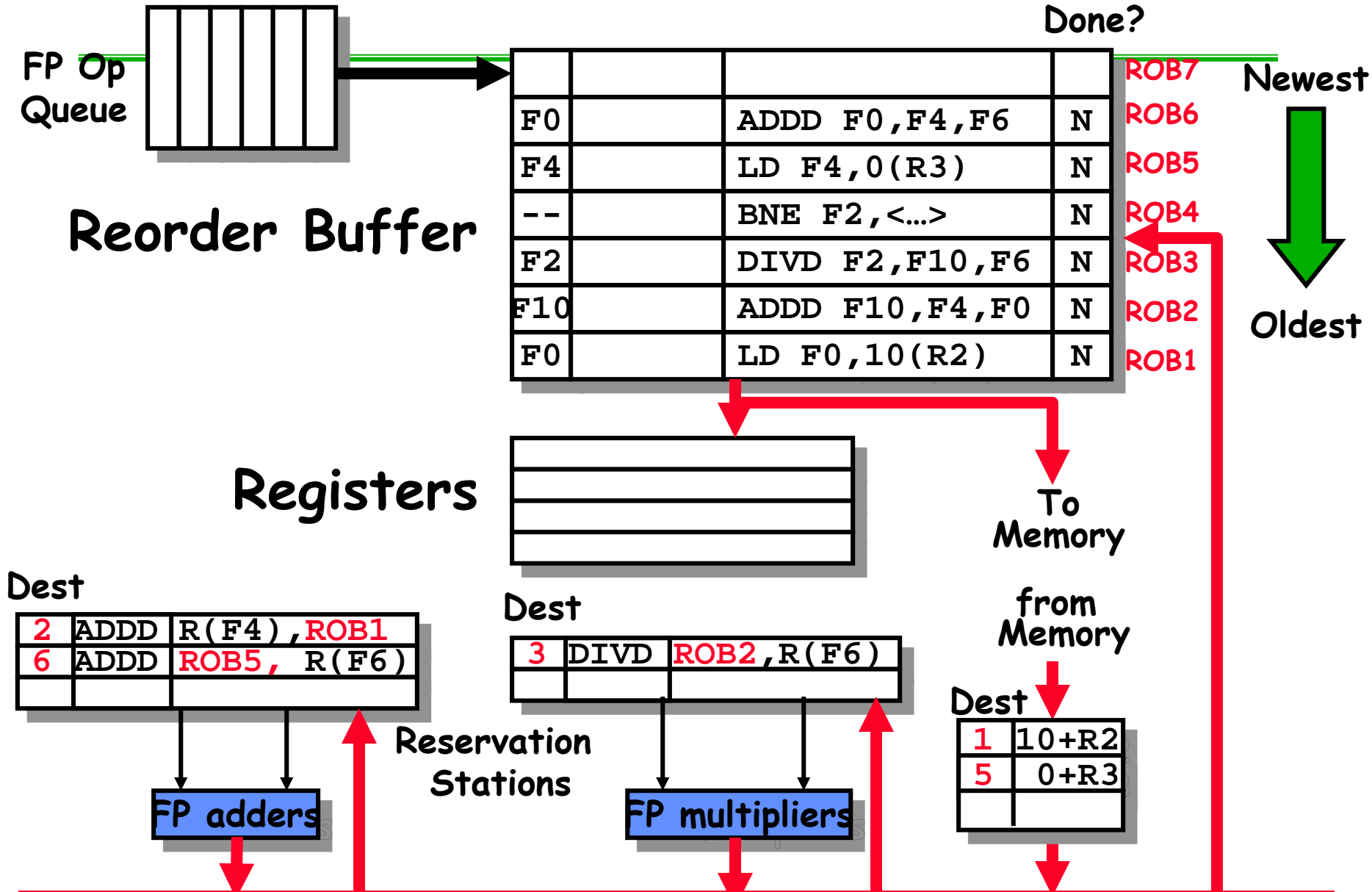
# Tomasulo With Reorder buffer:



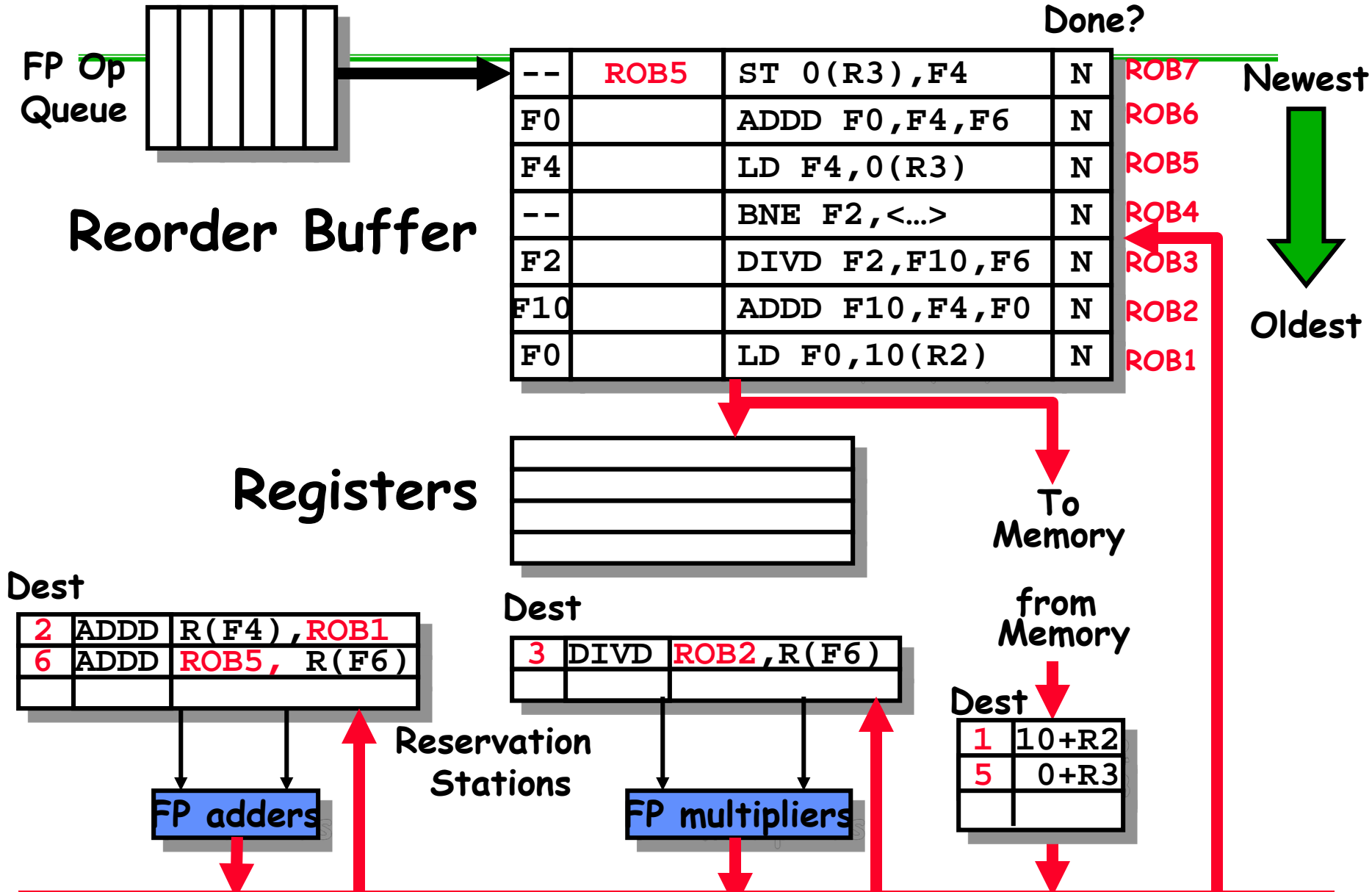
# Tomasulo With Reorder buffer:



# Tomasulo With Reorder buffer:

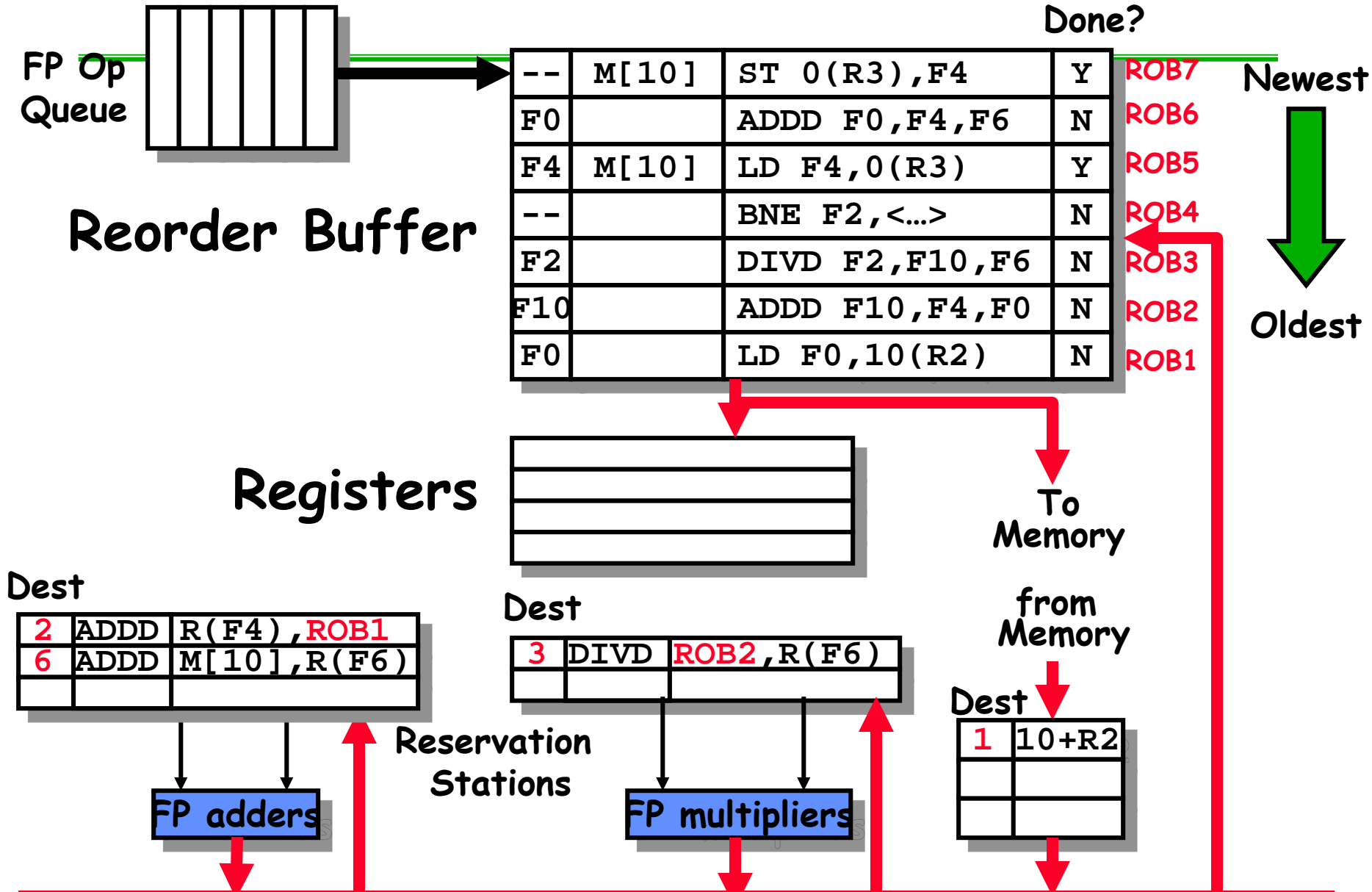


# Tomasulo With Reorder buffer:

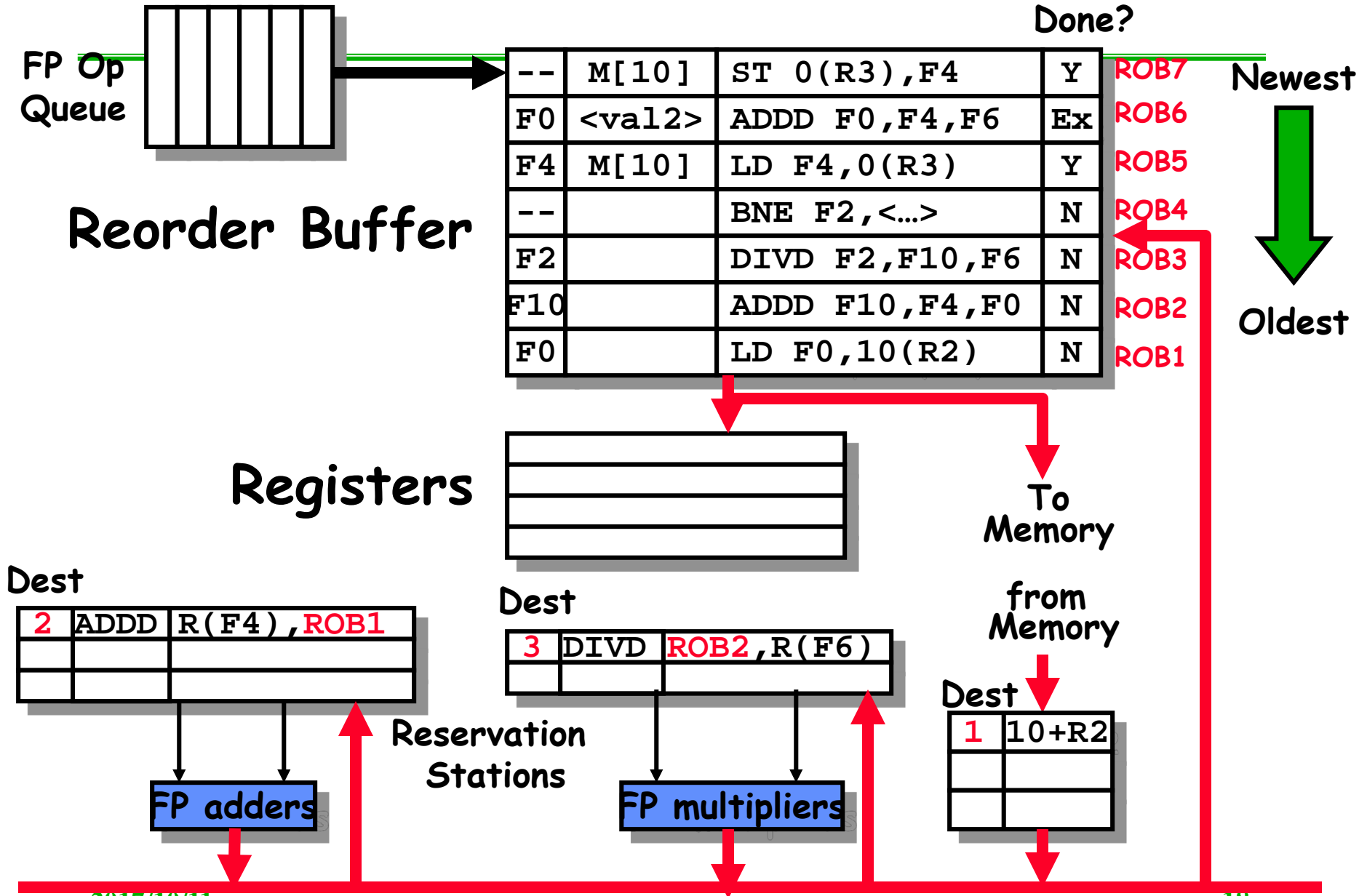




# Tomasulo With Reorder buffer:



# Tomasulo With Reorder buffer:



# Avoiding Memory Hazards

---

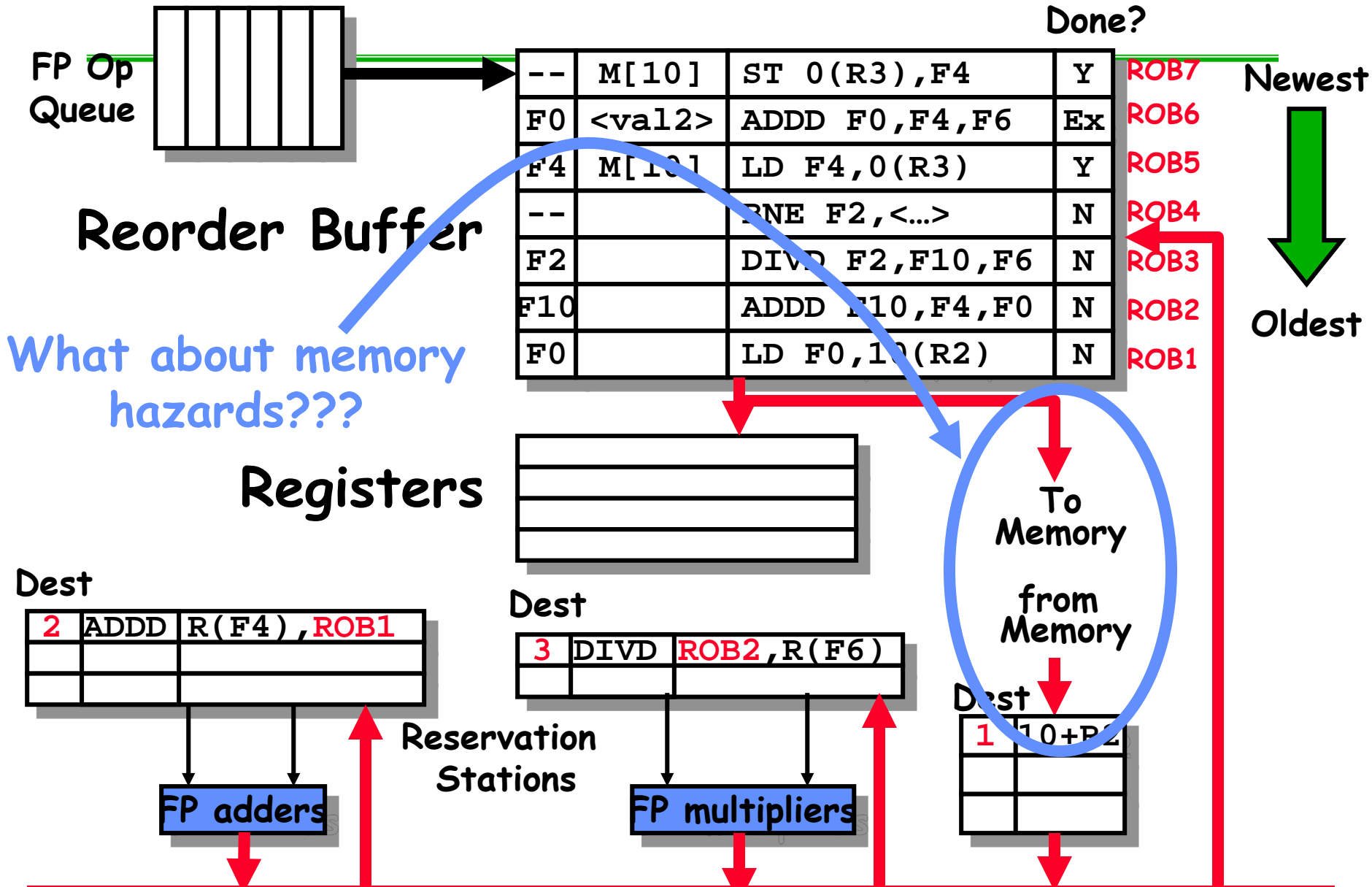
- **WAW and WAR hazards through memory are eliminated with speculation because actual updating of memory occurs in order, when a store is at head of the ROB, and hence, no earlier loads or stores can still be pending**
- **RAW hazards through memory are maintained by two restrictions:**
  1. **not allowing a load to initiate the second step of its execution if any active ROB entry occupied by a store has a Destination field that matches the value of the A field of the load, and**
  2. **maintaining the program order for the computation of an effective address of a load with respect to all earlier stores.**
- **these restrictions ensure that any load that accesses a memory location written to by an earlier store cannot perform the memory access until the store has written the data**

# 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 are in all new processors

# Tomasulo With Reorder buffer:



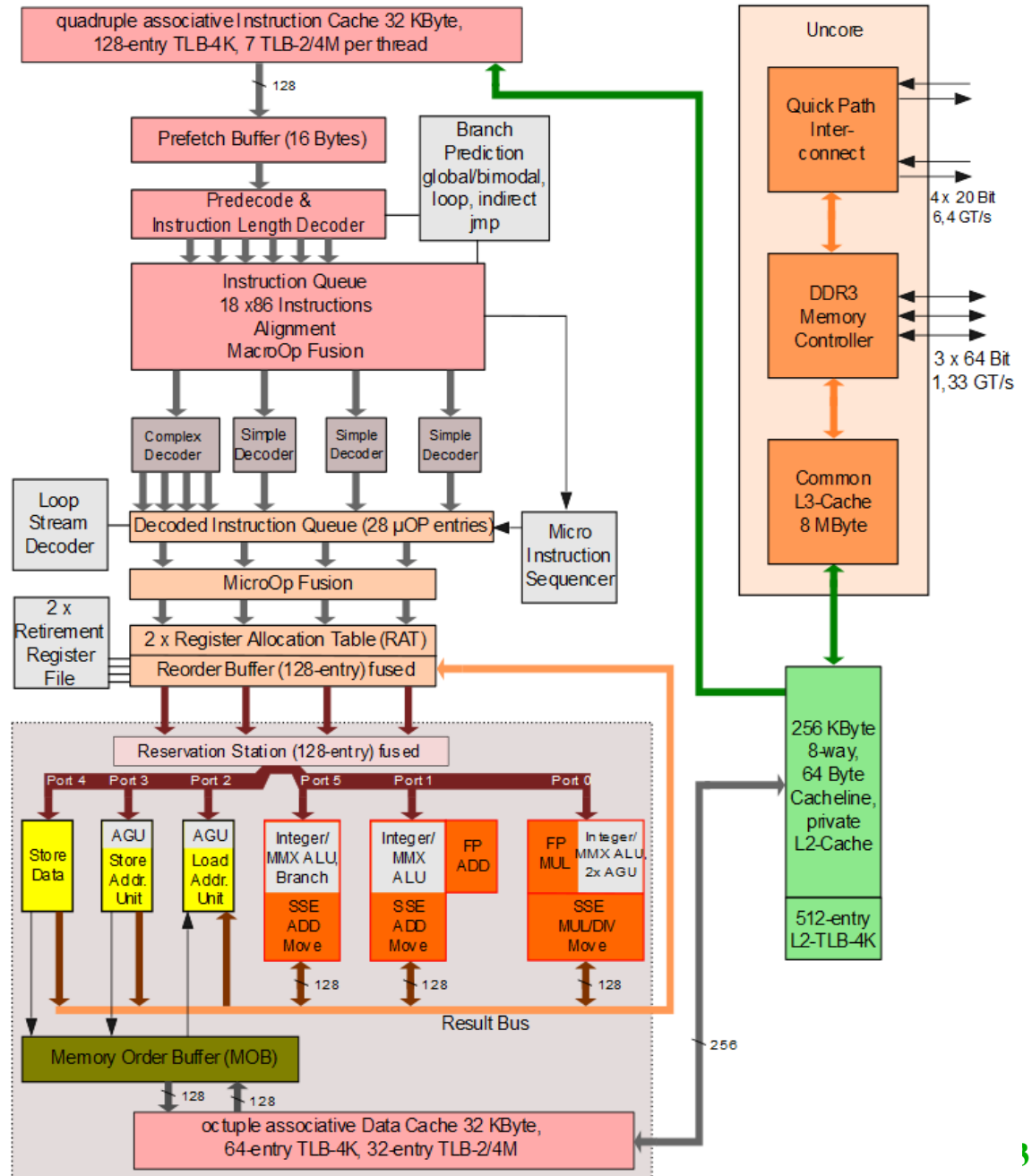
# Speculation: Register Renaming vs. ROB

---

- **Alternative to ROB is a larger physical set of registers combined with register renaming**
  - Extended registers replace function of both ROB and reservation stations
- **Instruction issue maps names of architectural registers to physical register numbers in extended register set**
  - On issue, allocates a new unused register for the destination (which avoids WAW and WAR hazards)
  - Speculation recovery easy because a physical register holding an instruction destination does not become the architectural register until the instruction commits
- **Most Out-of-Order processors today use extended registers with renaming**

# Reservation Station and ROB

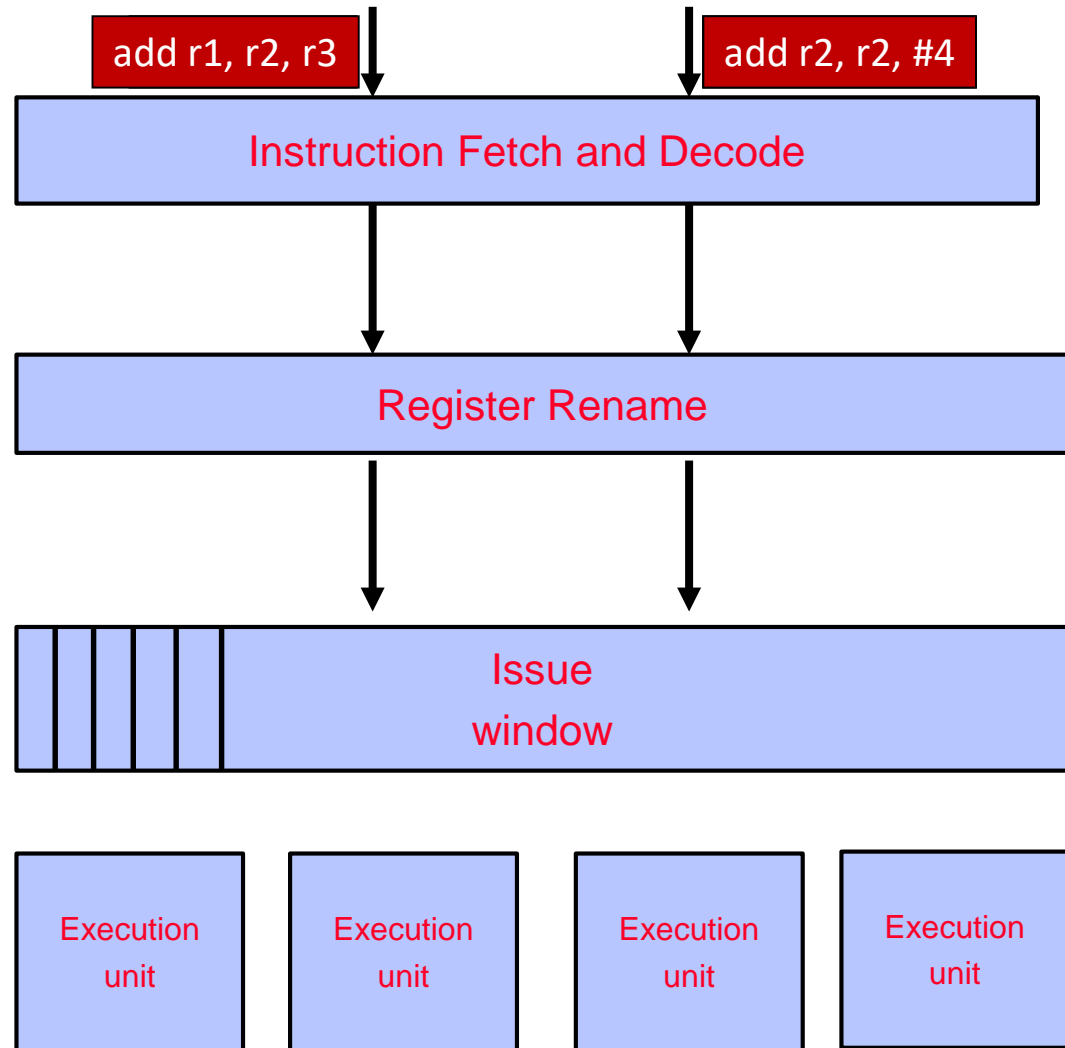
Dispatch 6 uops  
 Fetch/decode 4 instructions  
 128 reservation station entries



Intel Nehalem Microarchitecture

# Issue Window-Based OOO Processor Architecture

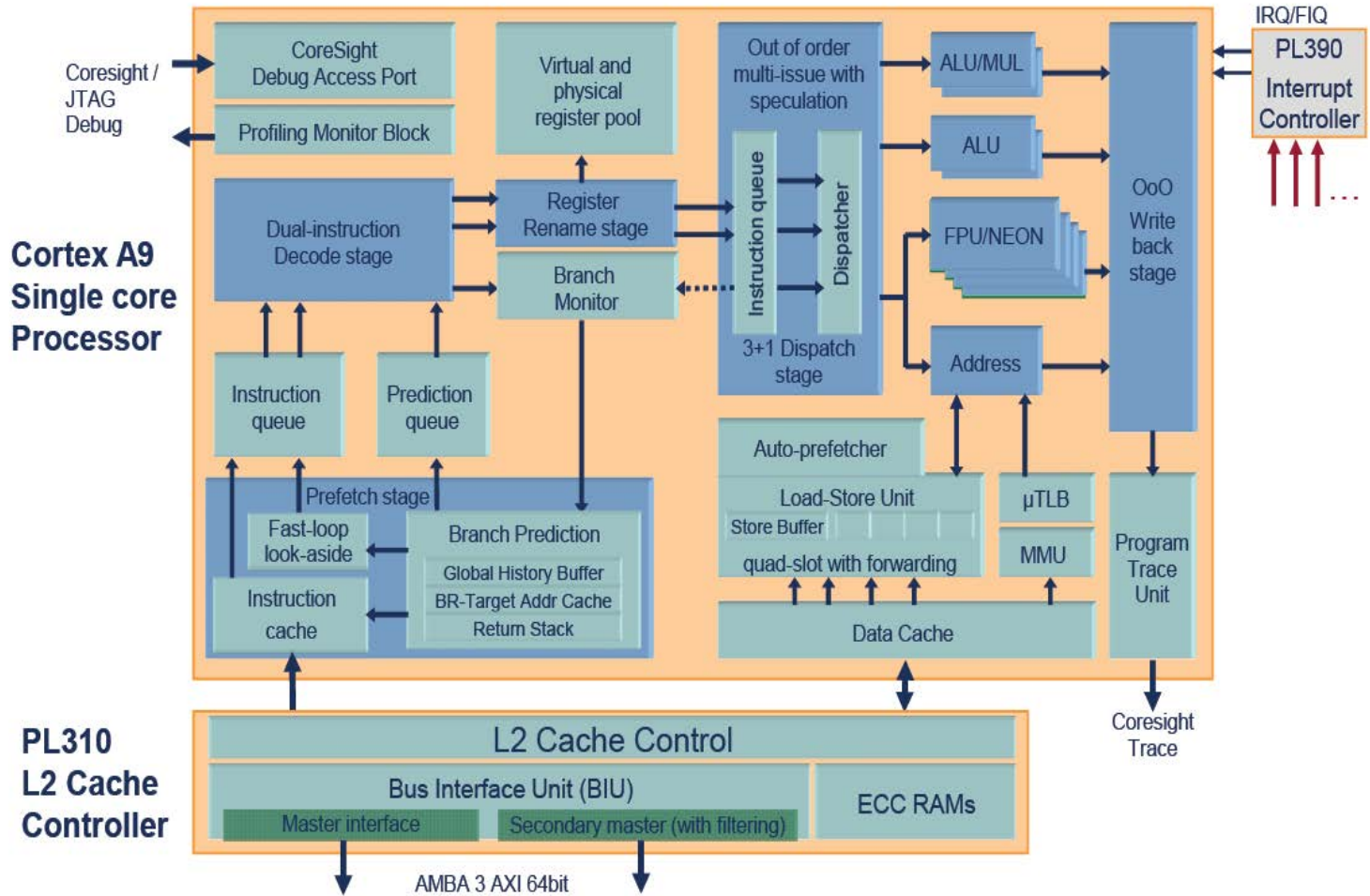
---





# Issue window based

Dispatch 4 instructions per clock cycle, fetch and decode 2 instructions



Cortex A9 Microarchitecture(single core variant)

# Performance beyond single thread 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**: process with own instructions 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

# Thread Level Parallelism (TLP)

---

- **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**

# Do both ILP and TLP?

---

- **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?**

# Simultaneous Multi-threading ...

## One thread, 8 units

Cycle M M FX FX FP FP BR CC

1	█							█
2	█	█					█	
3			█	█				
4								
5								
6								
7	█		█		█			
8		█		█				
9			█					

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

## Two threads, 8 units

Cycle M M FX FX FP FP BR CC

1	█	█	█					█
2	█	█	█			█	█	
3	█			█	█			
4	█	█				█		
5		█						█
6								
7	█		█	█	█	█		
8		█		█	█	█		
9	█	█		█		█		

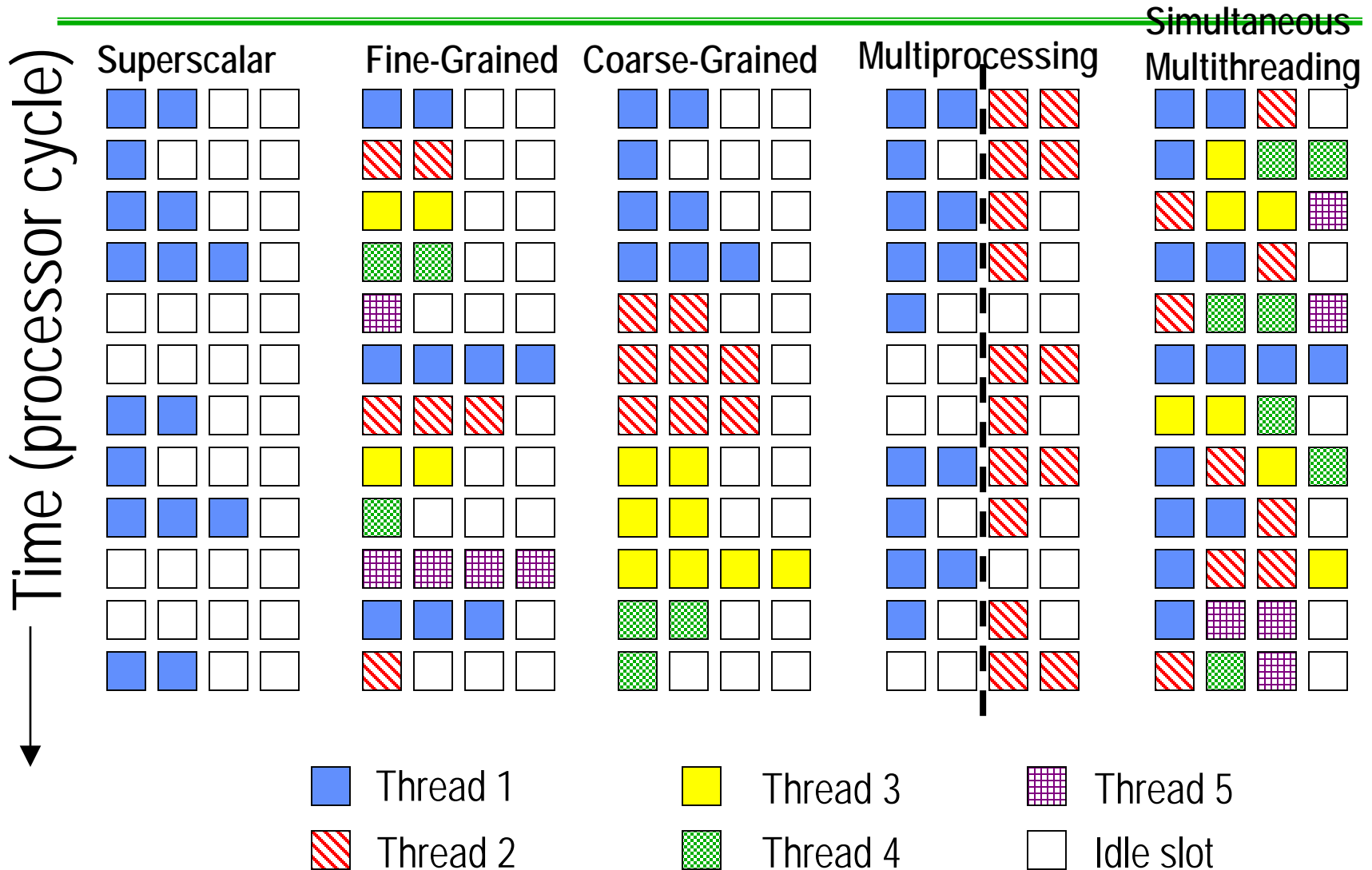
M = Load/Store, FX = Fixed Point, FP = Floating Point, BR = Branch, CC = Completion Codes

# Simultaneous Multithreading (SMT)

---

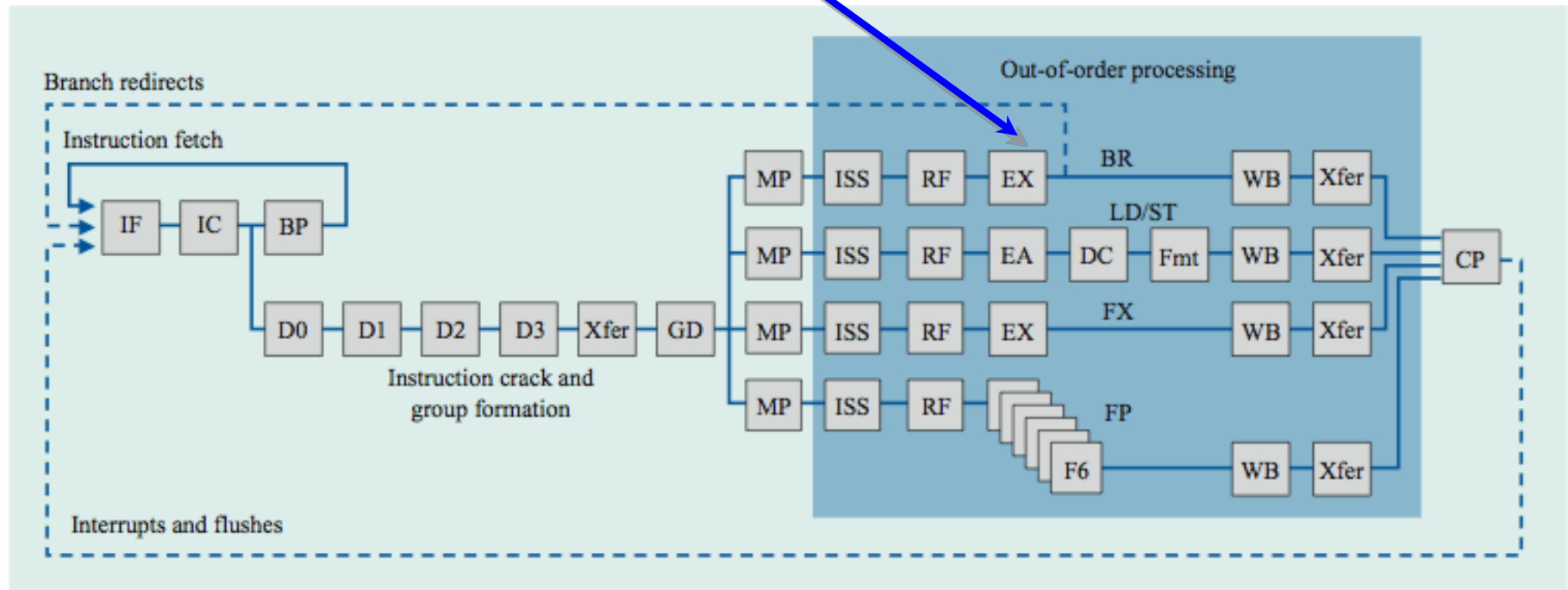
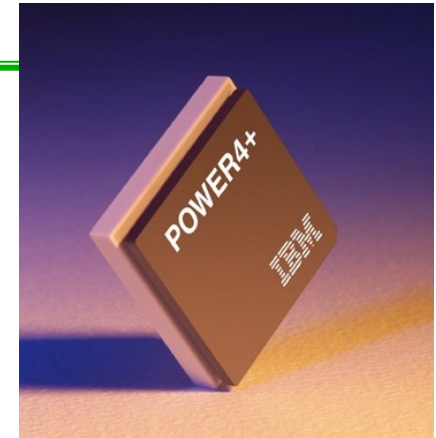
- **Simultaneous multithreading (SMT): insight that dynamically scheduled processor already has many HW mechanisms to support multithreading**
  - Large set of virtual registers that can be used to hold the register sets of independent threads
  - Register renaming provides unique register identifiers, so instructions from multiple threads can be mixed in datapath 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

# Multithreaded Categories



# Power 4

Single-threaded predecessor to Power 5. 8 execution units in out-of-order engine, each may issue an instruction each cycle.

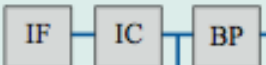




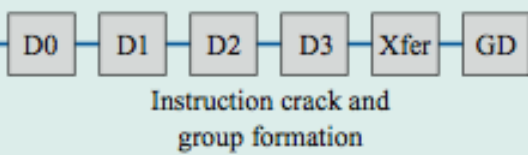
# Power 4

Branch redirects

Instruction fetch

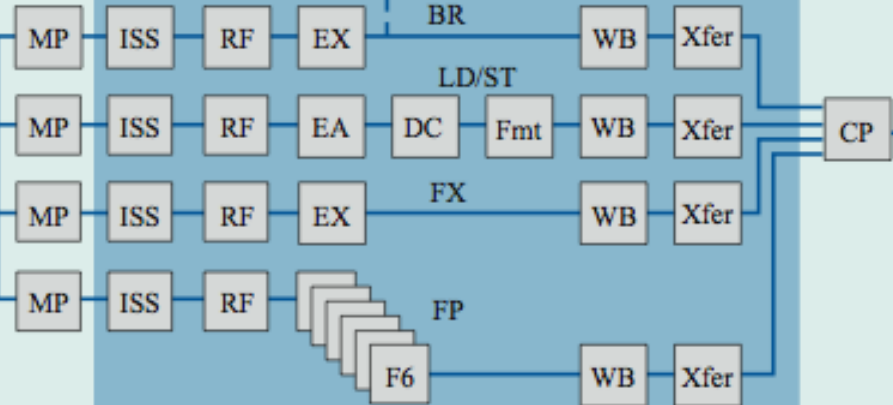


Instru  
cache



Interrupts and flushes

Out-of-order processing

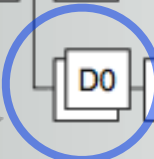
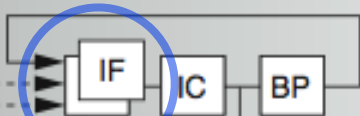


2 commits  
(architected  
register sets)

# Power 5

Branch redirects

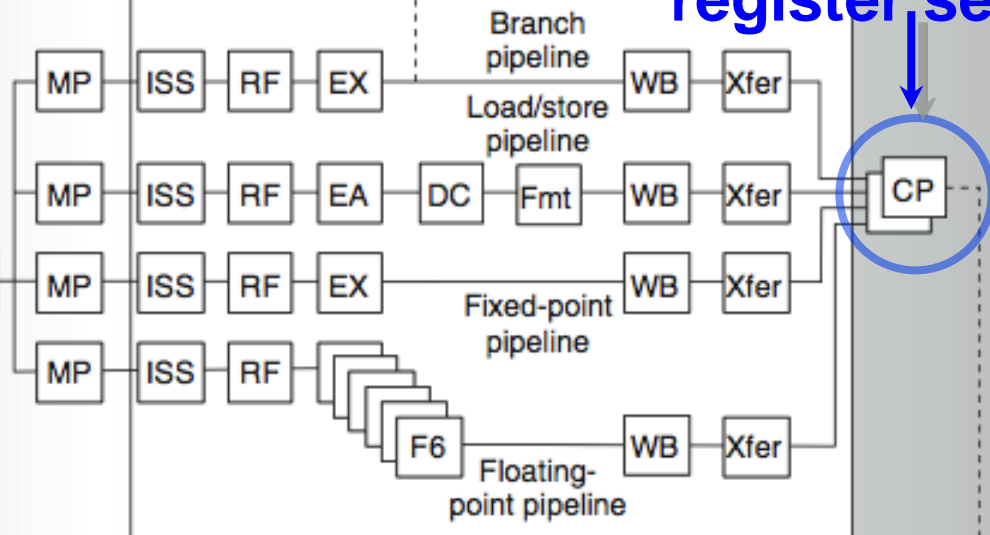
Instruction fetch



Group formation and  
instruction decode

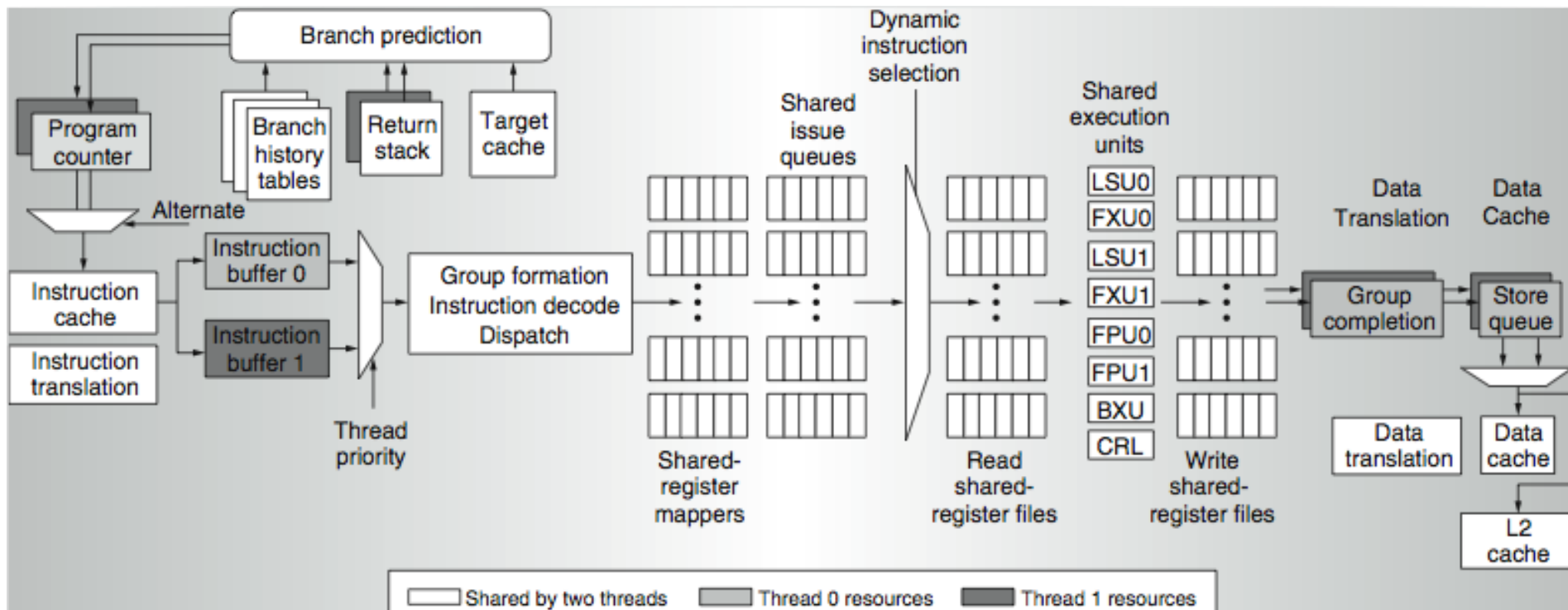
Interrupts and flushes

Out-of-order processing



2 fetch (PC),  
2 initial decodes

# Power 5 data flow ...



**Why only 2 threads? With 4, one of the shared resources (physical registers, cache, memory bandwidth) would be prone to bottleneck**

# Design Challenges in SMT

---

- **Since SMT makes sense only with fine-grained implementation, impact of fine-grained scheduling on single thread performance?**
  - A preferred thread approach sacrifices neither throughput nor single-thread performance?
  - Unfortunately, with a preferred thread, the processor is likely to sacrifice some throughput, when preferred thread stalls
- **Larger register file needed to hold multiple contexts**
- **Not affecting clock cycle time, especially in**
  - Instruction issue - more candidate instructions need to be considered
  - Instruction completion - choosing which instructions to commit may be challenging
- **Ensuring that cache and TLB conflicts generated by SMT do not degrade performance**

# In Conclusion ...

---

- **Interrupts and Exceptions either interrupt the current instruction or happen between instructions**
  - Possibly large quantities of state must be saved before interrupting
- **Machines with *precise exceptions* provide one single point in the program to restart execution**
  - All instructions before that point have completed
  - No instructions after or including that point have completed
- **Hardware techniques exist for precise exceptions even in the face of out-of-order execution!**
  - Important enabling factor for out-of-order execution