# Instruction Level Parallelism

# 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
- Helps cache misses as well
- Lasting Contributions
  - Dynamic scheduling
  - Register renaming
- 360/91 descendants are Pentium 4, Power 5, AMD Athlon/Opteron, …

# Outline

- Speculation
- Speculative Tomasulo Example
- Memory Aliases
- Exceptions
- VLIW
- Increasing instruction bandwidth
- Register Renaming vs. Reorder Buffer
- Value Prediction

# Speculation for 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 $\Rightarrow$ only fetches and issues instructions
- Essentially a data flow execution model: Operations execute as soon as their operands are available

# Speculation for 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 architectured registers like RS

# Reorder Buffer Entry

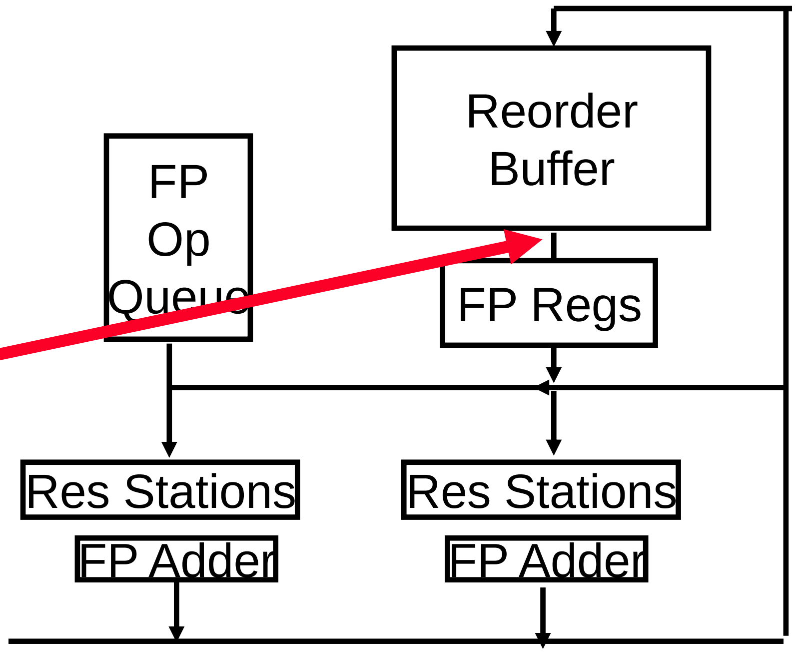
- Each entry in the ROB contains four fields:

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 ⇒ more registers like RS
  - Tag results with ROB buffer number instead of reservation station
- Instructions commit ⇒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

3. Write result—finish execution (WB)

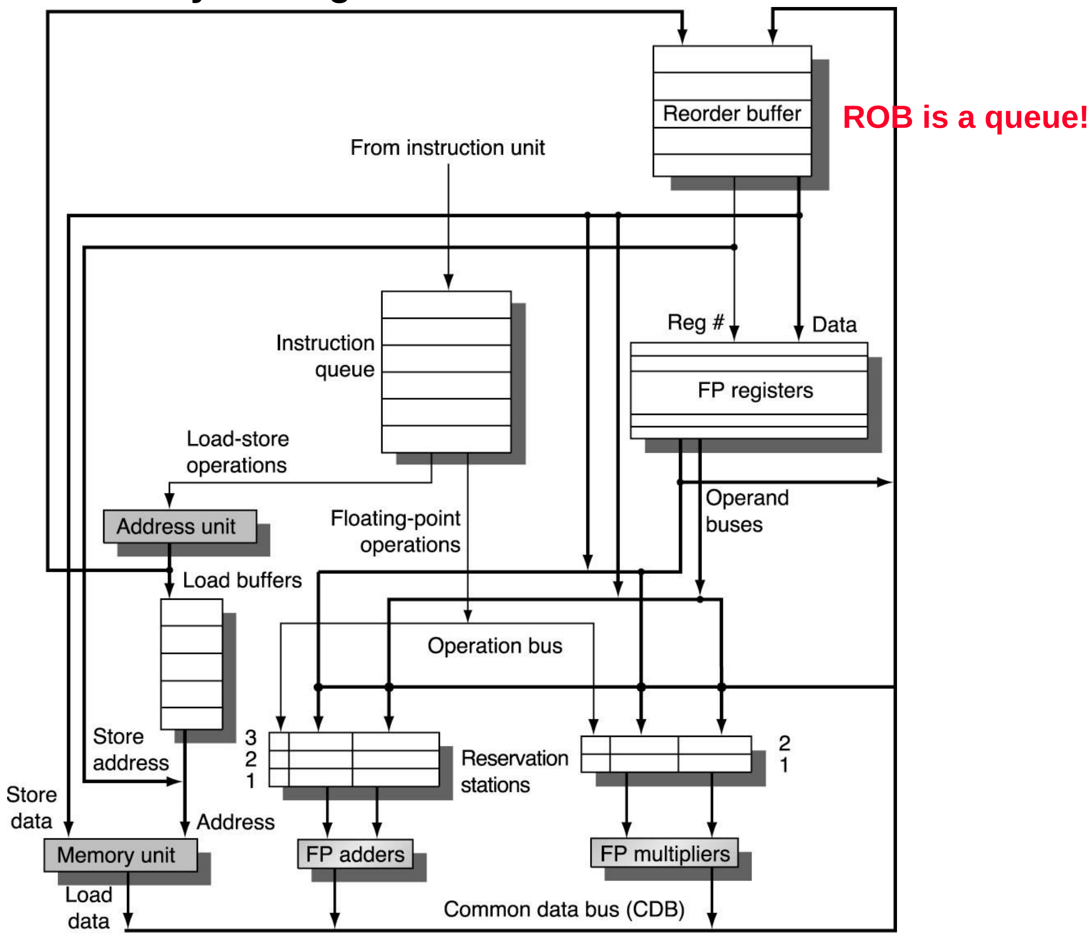
   Write on Common Data Bus to all awaiting FUs & reorder buffer; mark reservation station available.

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 Hardware with Speculation

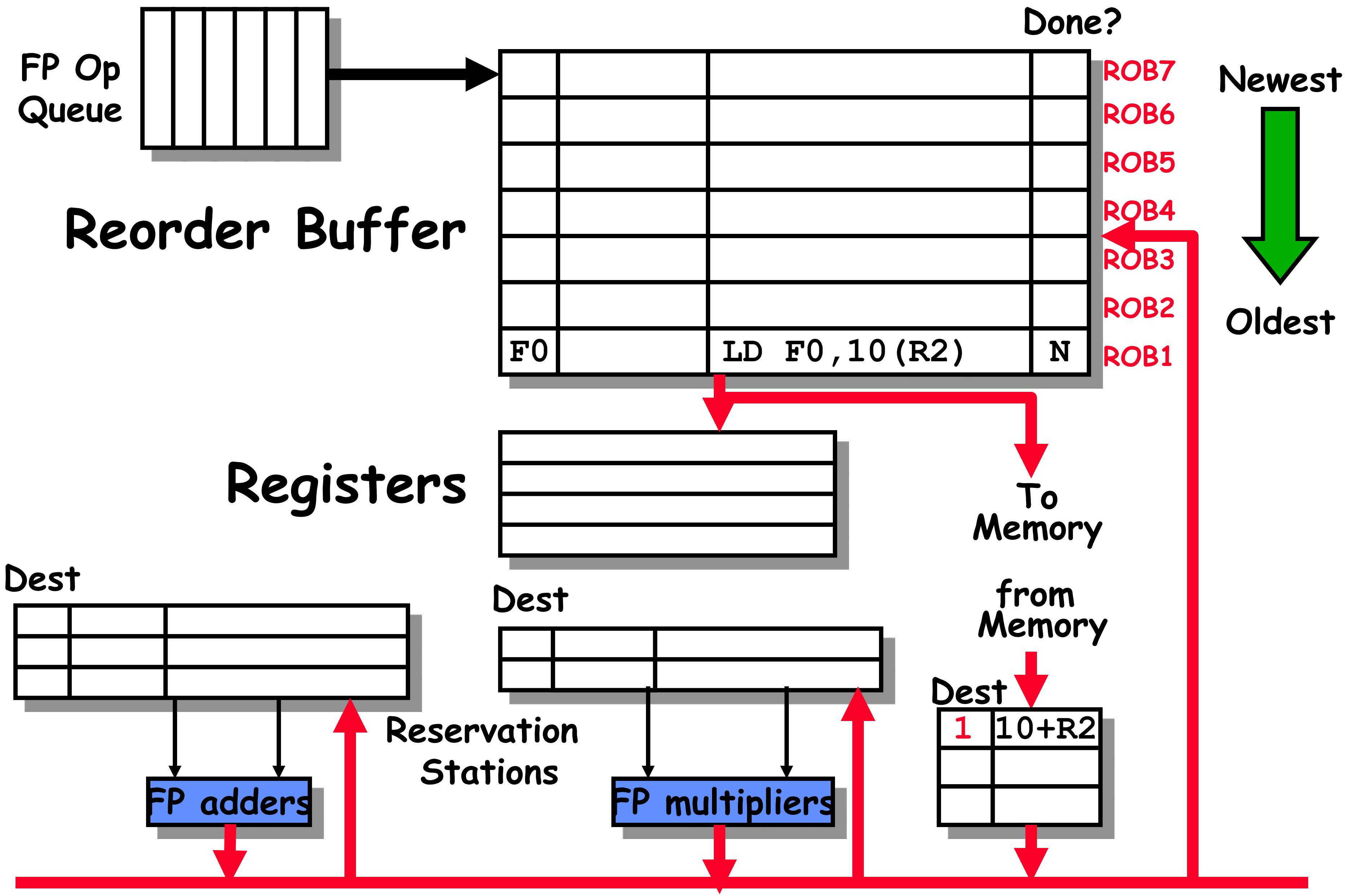
**Basic structure of MIPS floating-point unit based on Tomasulo and extended to handle speculation: ROB is added and store buffer in original Tomasulo is eliminated as its functionality is integrated into the ROB**



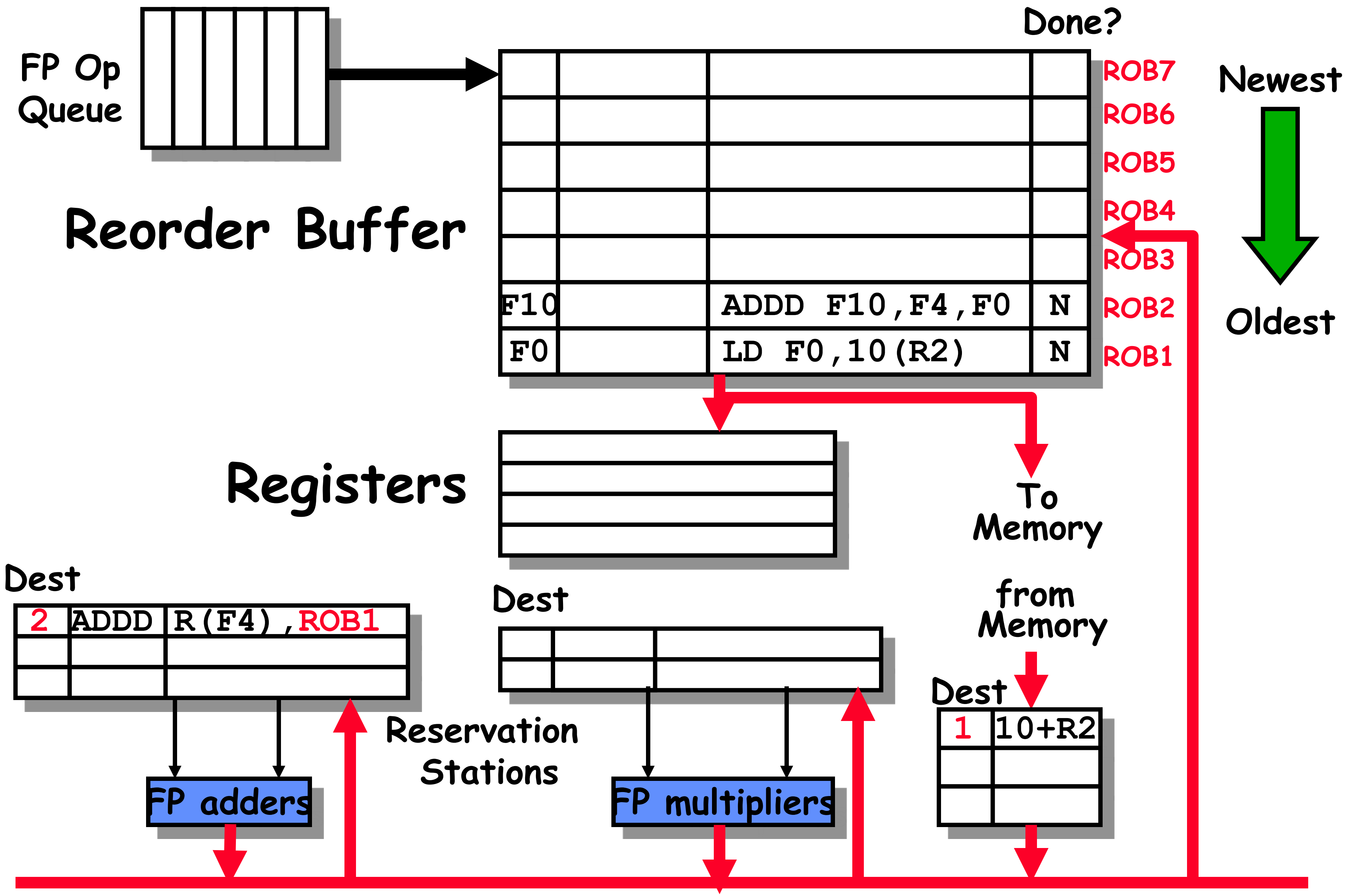
# Speculative Tomasulo Example

```
        LD      F0   10   R2
        ADDD    F10  F4   F0
        DIVD    F2   F10  F6
        BNEZ    F2   Exit
        LD      F4   0    R3
        ADDD    F0   F4   F9
        SD      F4   0    R3
        …
Exit:
```

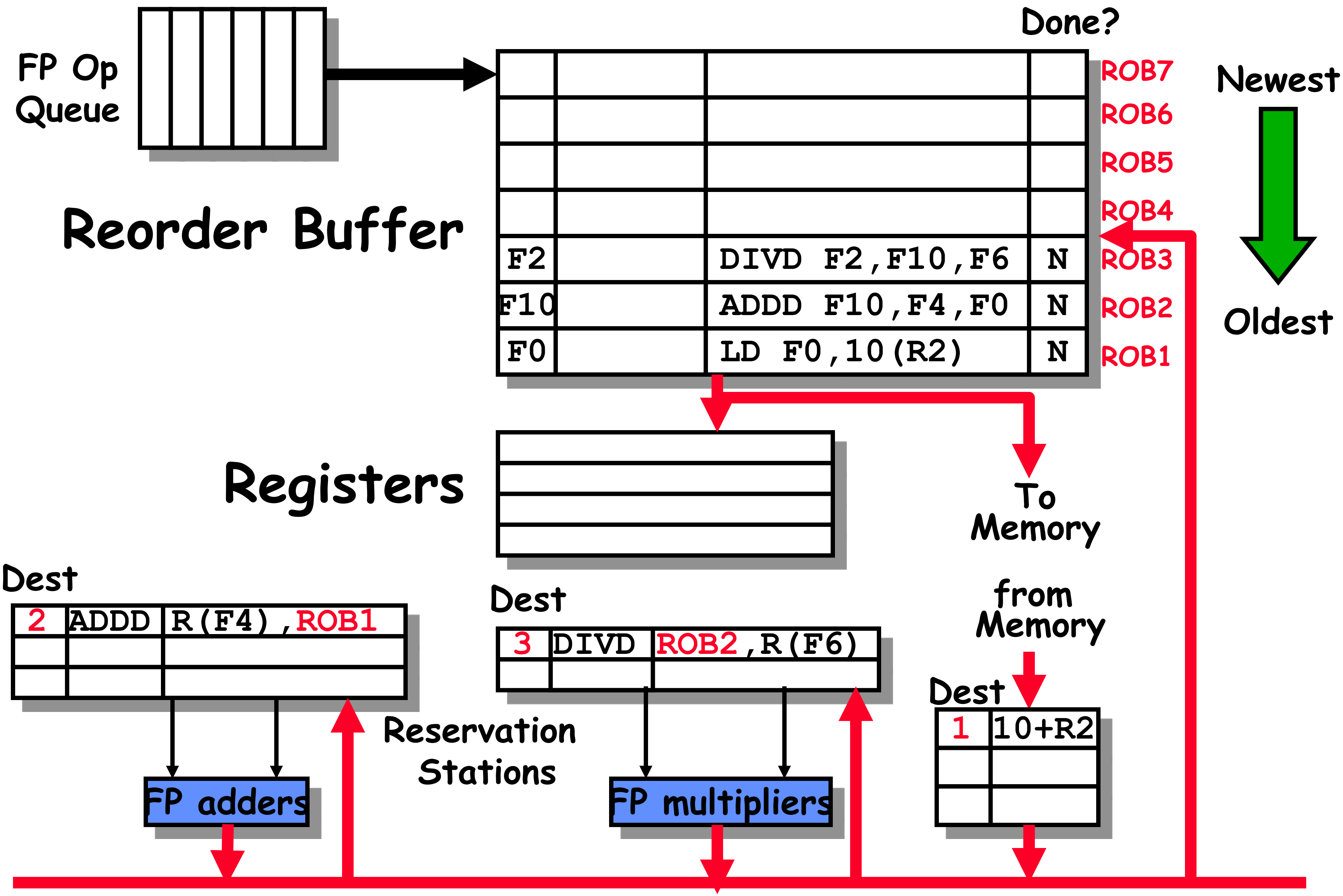
# Tomasulo With Reorder buffer:



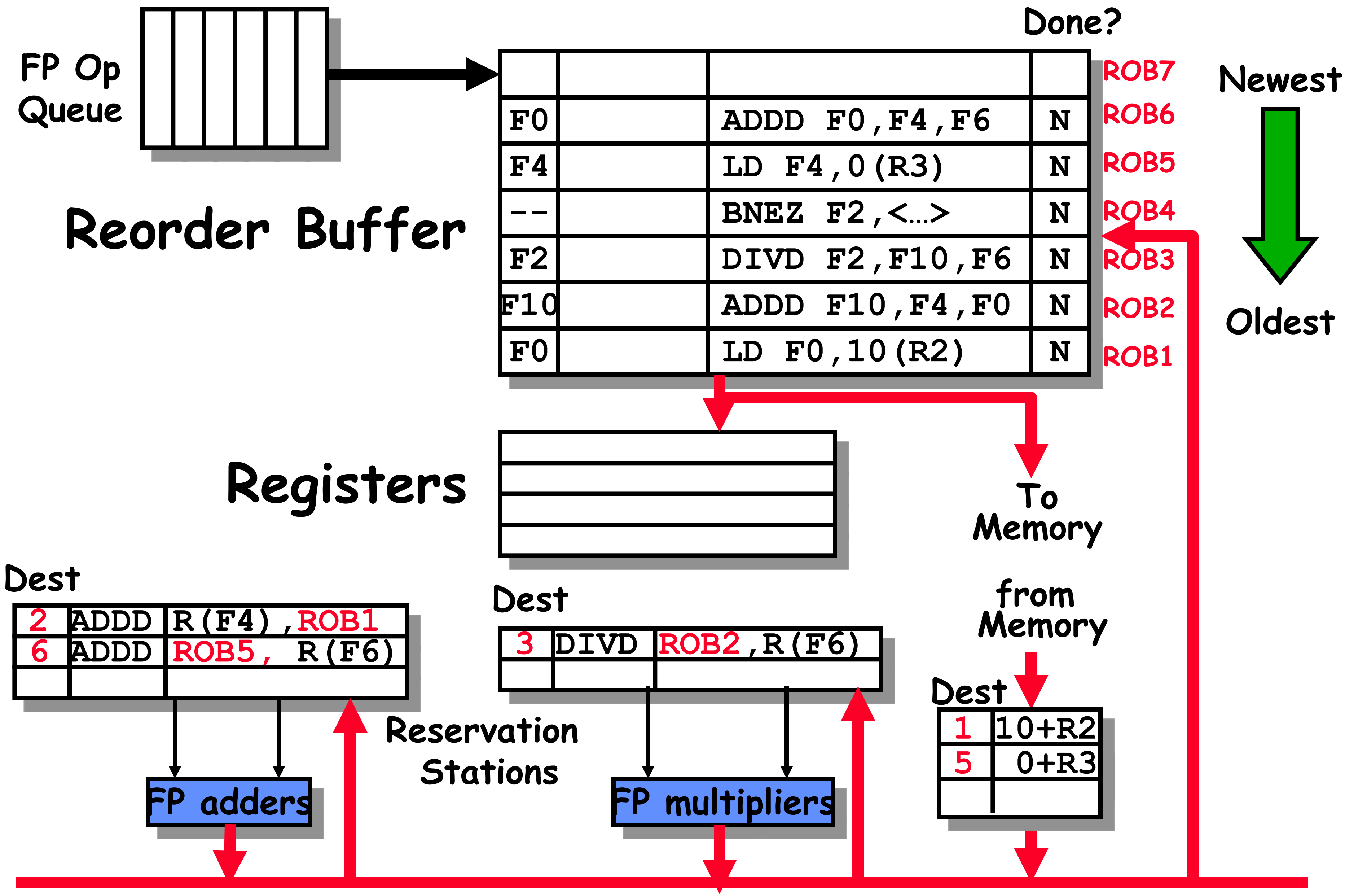
# Tomasulo With Reorder buffer:



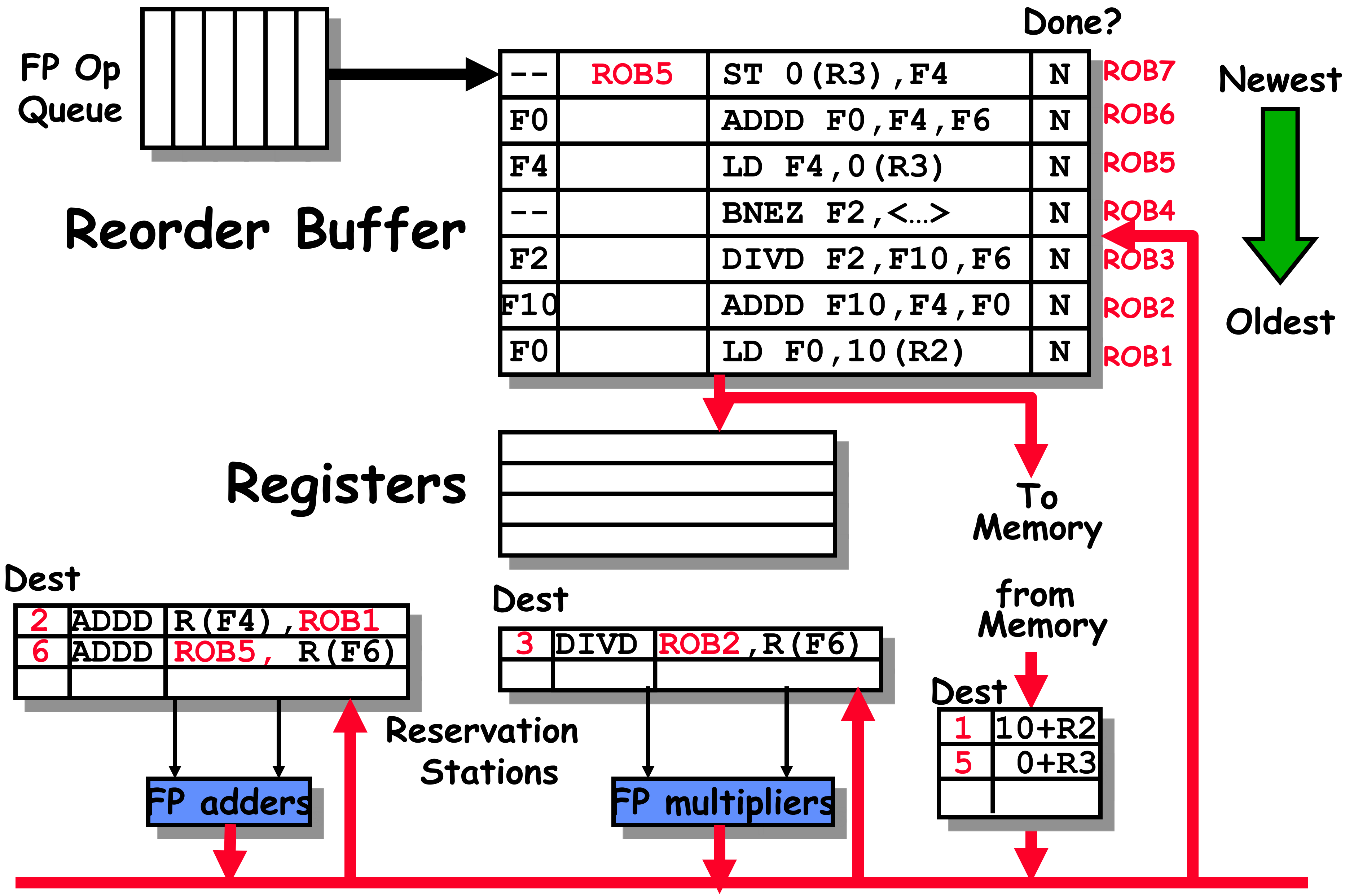
# Tomasulo With Reorder buffer:



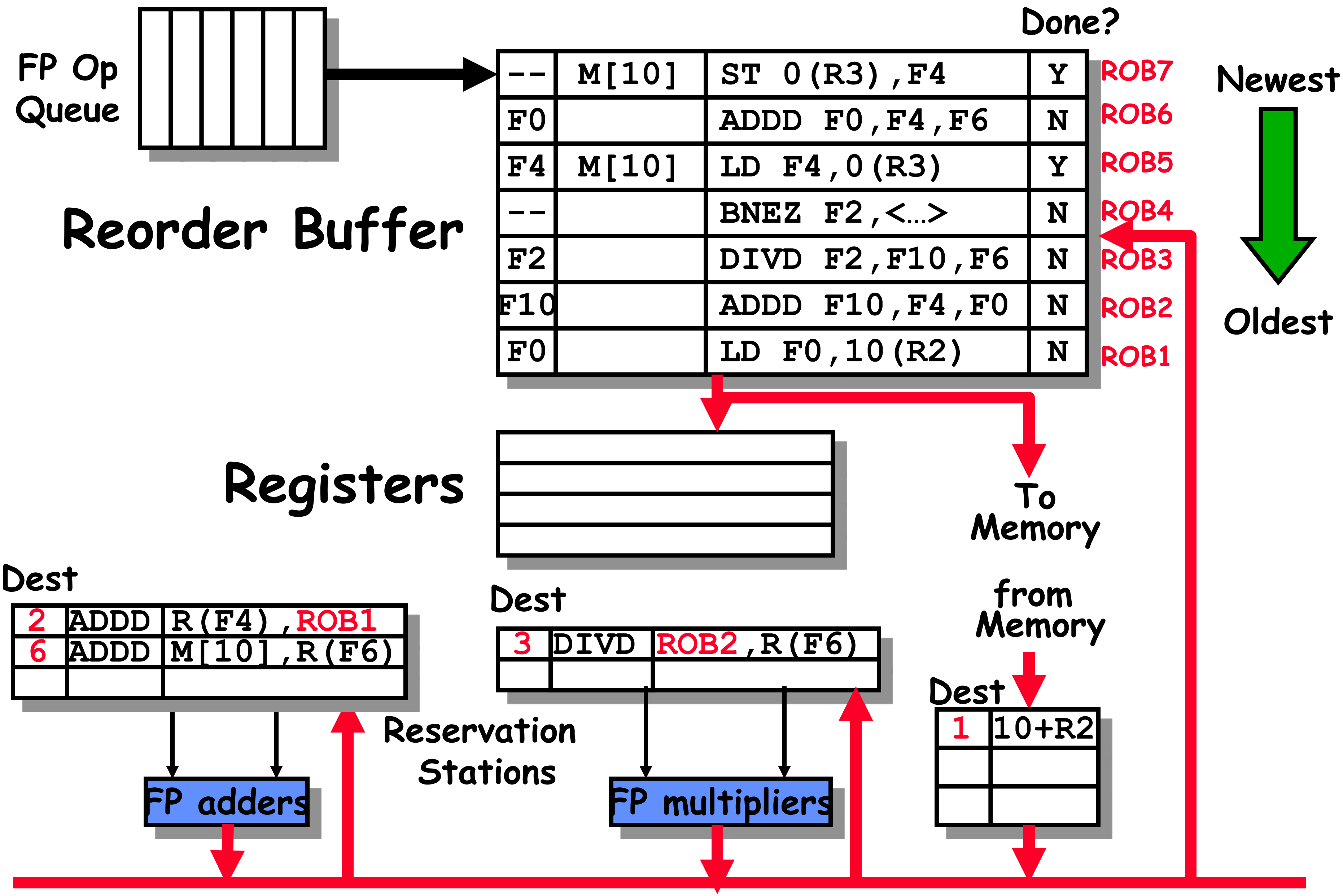
# Tomasulo With Reorder buffer:



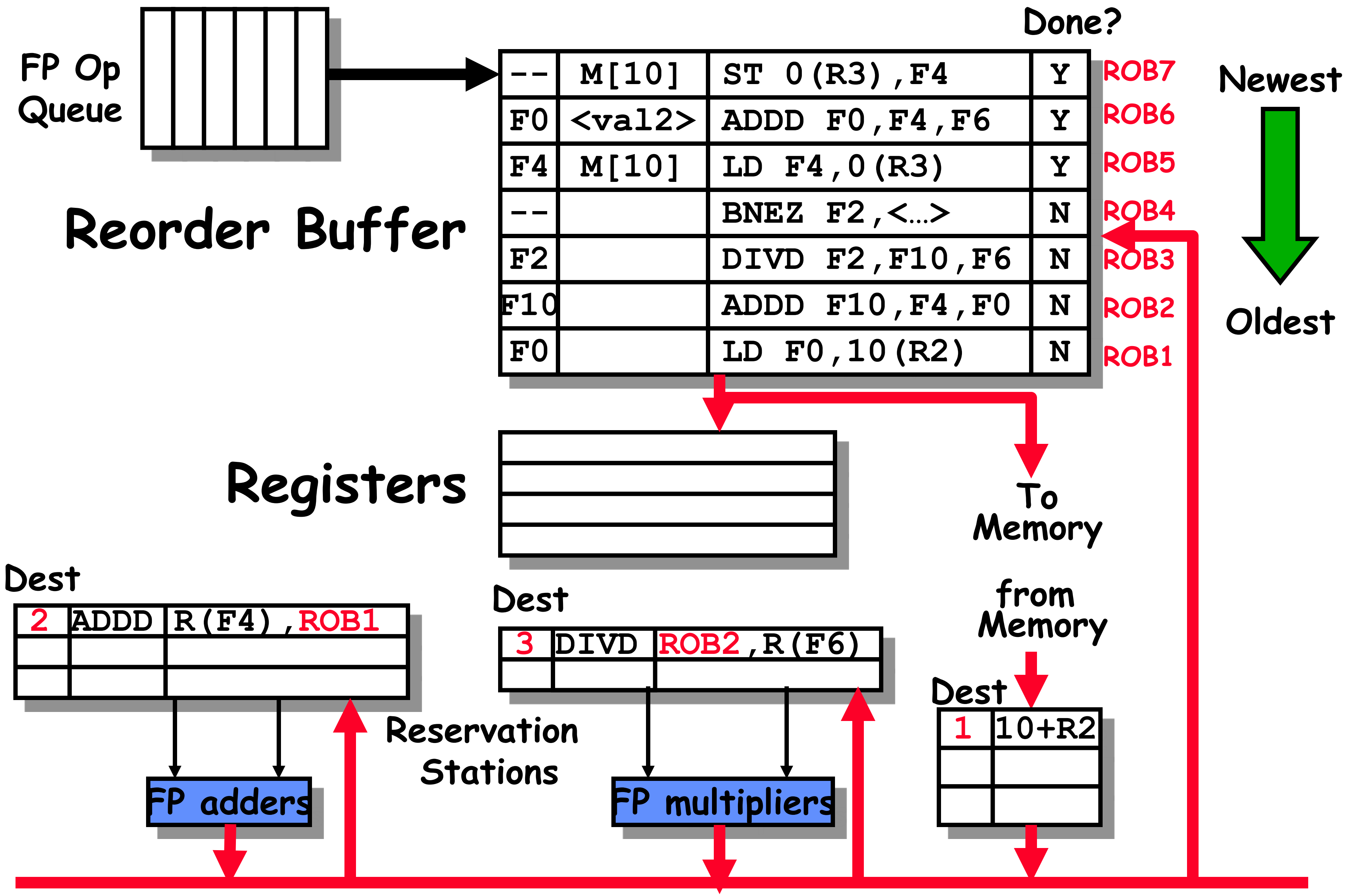
# Tomasulo With Reorder buffer:



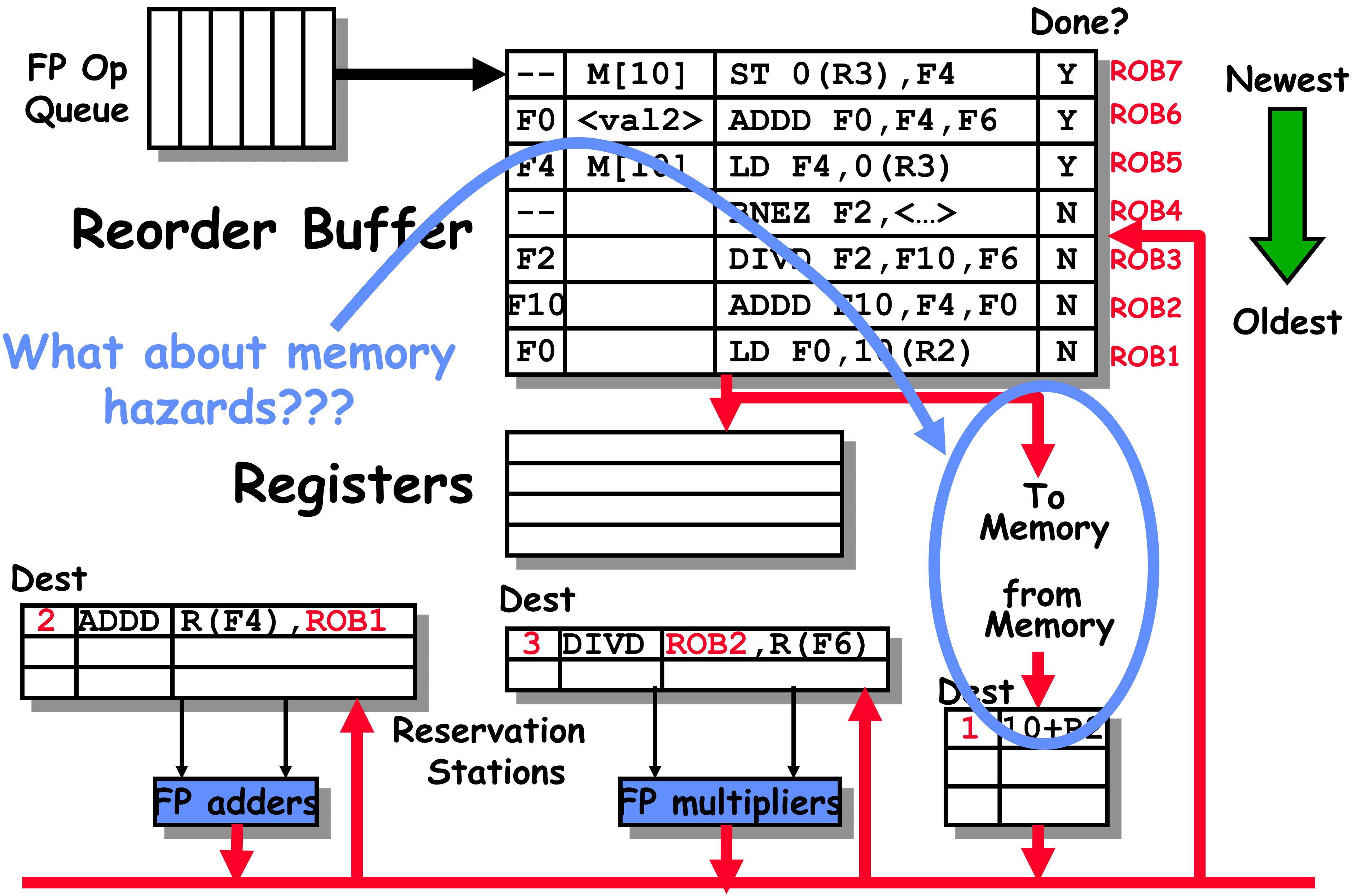
# Tomasulo With Reorder buffer:



# Tomasulo With Reorder buffer:



# Tomasulo With Reorder buffer:



# Notes

- If a branch is mispredicted, recovery is done by flushing the ROB of all entries that appear after the mispredicted branch
  - entries before the branch are allowed to continue
  - restart the fetch at the correct branch successor
- When an instruction commits or is flushed from the ROB then the corresponding slots become available for subsequent instructions

# 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”
- 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 used in all new processors

# Floating Point Pipeline

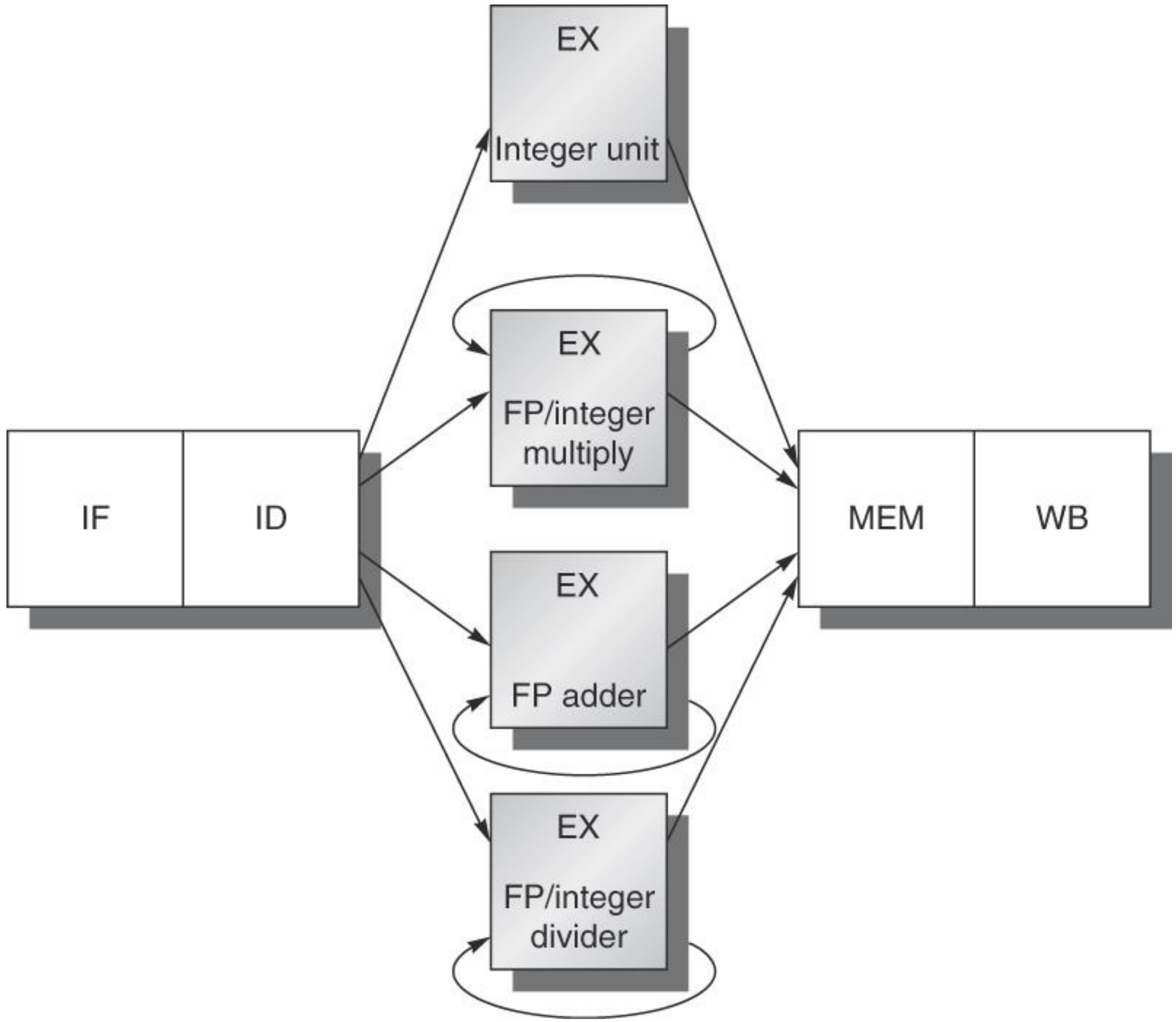



**Figure C.33 The MIPS pipeline with three additional unpipelined, floating-point, functional units. Because only one instruction issues on every clock cycle, all instructions go through the standard pipeline for integer operations. The FP operations simply loop when they reach the EX stage. After they have finished the EX stage, they proceed to MEM and WB to complete execution.**

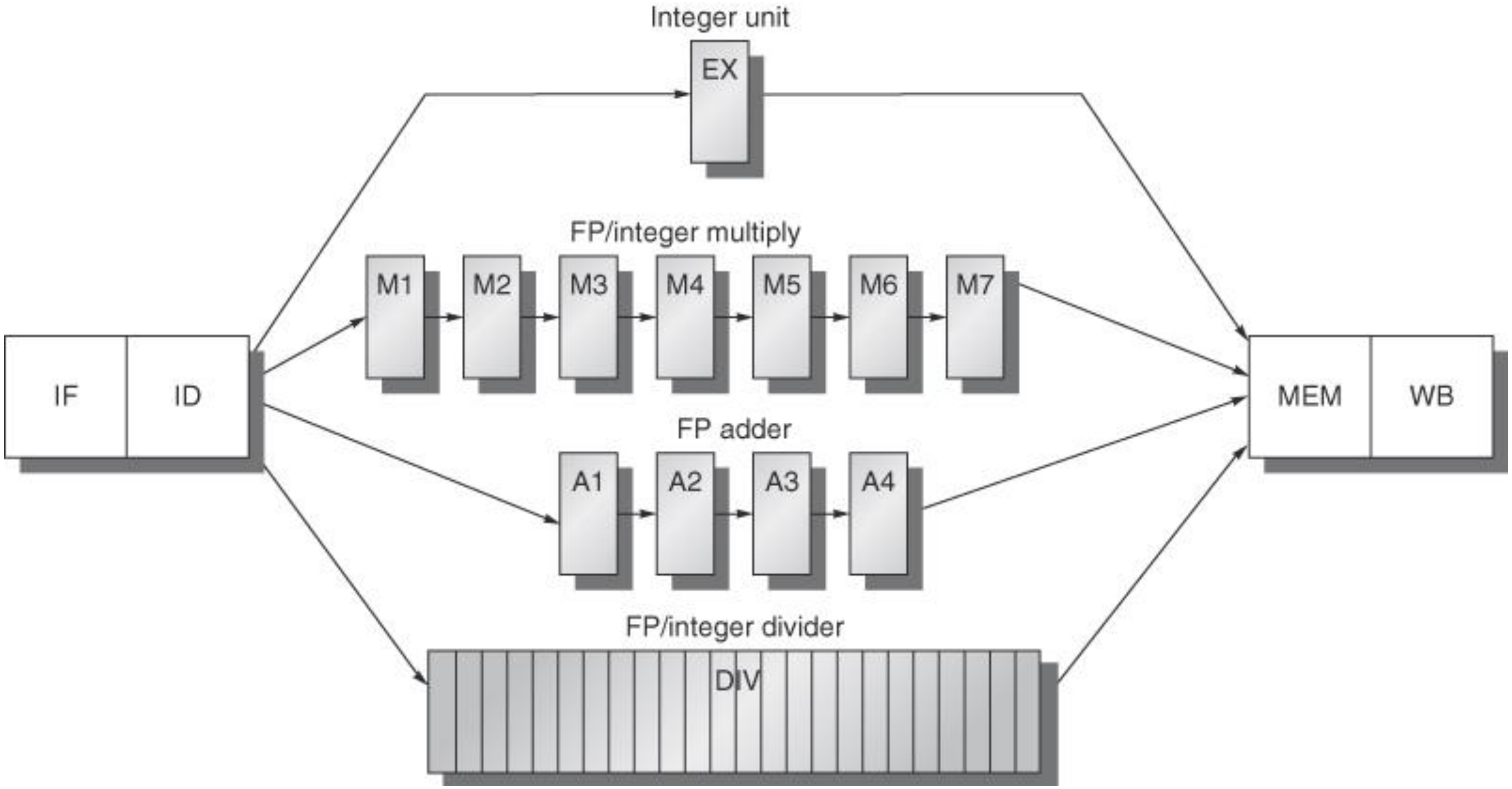



**Figure C.35 A pipeline that supports multiple outstanding FP operations. The FP multiplier and adder are fully pipelined and have a depth of seven and four stages, respectively. The FP divider is not pipelined, but requires 24 clock cycles to complete. The latency in instructions between the issue of an FP operation and the use of the result of that operation without incurring a RAW stall is determined by the number of cycles spent in the execution stages. For example, the fourth instruction after an FP add can use the result of the FP add. For integer ALU operations, the depth of the execution pipeline is always one and the next instruction can use the results.**

| | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| MUL.D | IF | ID | *M1* | M2 | M3 | M4 | M5 | M6 | M7 | MEM | WB |
| ADD.D | | IF | ID | *A1* | A2 | A3 | **A4** | **MEM** | WB | | |
| L.D | | | IF | ID | *EX* | **MEM** | WB | | | | |
| S.D | | | | IF | ID | *EX* | *MEM* | WB | | | |

**Figure C.36 The pipeline timing of a set of independent FP operations.** The stages in italics show where data are needed, while the stages in bold show where a result is available. The ".D" extension on the instruction mnemonic indicates double-precision (64-bit) floating-point operations. FP loads and stores use a 64-bit path to memory so that the pipelining timing is just like an integer load or store.

1. Because the divide unit is not fully pipelined, structural hazards can occur.
2. Because the instructions have varying running times, the number of register writes required in a cycle can be larger than 1.
3. Write after write (WAW) hazards are possible, since instructions no longer reach WB in order. Note that write after read (WAR) hazards are not possible, since the register reads always occur in ID.
4. Instructions can complete in a different order than they were issued, causing problems with exceptions; we deal with this in the next subsection.
5. Because of longer latency of operations, stalls for RAW hazards will be more frequent.

| | | Clock cycle number | | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Instruction | | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 | 16 | 17 |
| L.D | F4,0(R2) | IF | ID | EX | MEM | WB | | | | | | | | | | | | |
| MUL.D | F0,F4,F6 | | IF | ID | Stall | M1 | M2 | M3 | M4 | M5 | M6 | M7 | MEM | WB | | | | |
| ADD.D | F2,F0,F8 | | | IF | Stall | ID | Stall | Stall | Stall | Stall | Stall | Stall | A1 | A2 | A3 | A4 | MEM | WB |
| S.D | F2,0(R2) | | | | | IF | Stall | Stall | Stall | Stall | Stall | Stall | ID | EX | Stall | Stall | Stall | MEM |

**Figure C.37 A typical FP code sequence showing the stalls arising from RAW hazards.** The longer pipeline substantially raises the frequency of stalls versus the shallower integer pipeline. Each instruction in this sequence is dependent on the previous and proceeds as soon as data are available, which assumes the pipeline has full bypassing and forwarding. The S.D must be stalled an extra cycle so that its MEM does not conflict with the ADD.D. Extra hardware could easily handle this case.

| | Clock cycle number | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| Instruction | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 |
| MUL.D F0,F4,F6 | IF | ID | M1 | M2 | M3 | M4 | M5 | M6 | M7 | MEM | WB |
| ... | | IF | ID | EX | MEM | WB | | | | | |
| ... | | | IF | ID | EX | MEM | WB | | | | |
| ADD.D F2,F4,F6 | | | | IF | ID | A1 | A2 | A3 | A4 | MEM | WB |
| ... | | | | | IF | ID | EX | MEM | WB | | |
| ... | | | | | | IF | ID | EX | MEM | WB | |
| L.D F2,0(R2) | | | | | | | IF | ID | EX | MEM | WB |

**Figure C.38 Three instructions want to perform a write-back to the FP register file simultaneously, as shown in clock cycle 11.** This is *not* the worst case, since an earlier divide in the FP unit could also finish on the same clock. Note that although the MUL.D, ADD.D, and L.D all are in the MEM stage in clock cycle 10, only the L.D actually uses the memory, so no structural hazard exists for MEM.

# Getting CPI < 1:
# Issuing Multiple Instructions (Ops)/Cycle

# Getting CPI < 1: Issuing Multiple Instructions (Ops)/Cycle

- Vector Processing: Explicit coding of independent loops as operations on large vectors of numbers
  - Multimedia instructions being added to many processors
- Superscalar: varying no. instructions/cycle (1 to 8), scheduled by compiler or by HW (Tomasulo)
  - IBM PowerPC, Sun UltraSparc, DEC Alpha, Pentium 4
- (Very) Long Instruction Words (V)LIW: fixed number of instructions (4-16) scheduled by the compiler; put ops into wide templates
  - Intel Architecture-64 (IA-64) 64-bit address
- Parallel processing:
  - Intel Core 2 Duo

* Anticipated success of multiple instructions lead to Instructions Per Clock_cycle (IPC) vs. CPI