Lecture 17: Instruction Level Parallelism
-- Hardware Speculation and VLIW (Static Superscalar)

CSCE 513 Computer Architecture

Department of Computer Science and Engineering

Yonghong Yan

yanyh@cse.sc.edu

https://passlab.github.io/CSCE513

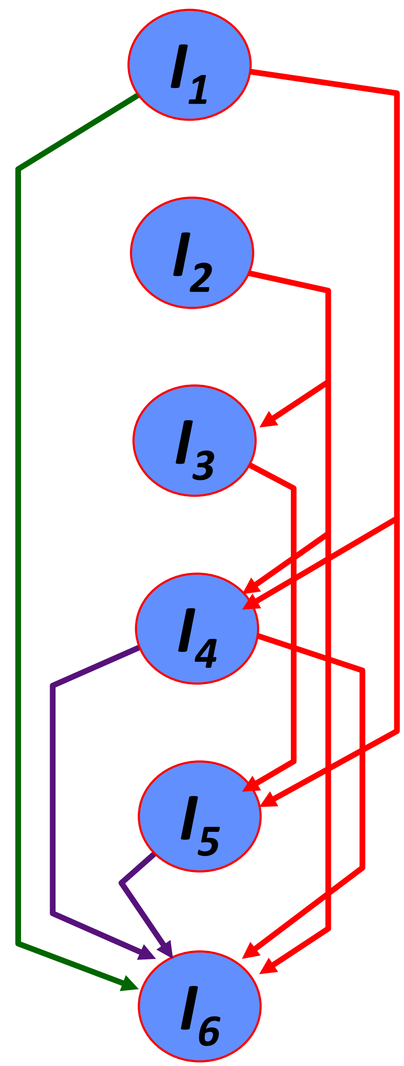
Topics for Instruction Level Parallelism

- 5-stage Pipeline Extension, ILP Introduction, Compiler Techniques, and Branch Prediction
  - C.5, C.6
  - 3.1, 3.2
  - ~~Branch Prediction, C.2, 3.3~~
- Dynamic Scheduling (OOO)
  - 3.4, 3.5
- Hardware Speculation and Static Superscalar/VLIW
  - 3.6, 3.7
- Dynamic Superscalar, Advanced Techniques, ARM Cortex-A53, and Intel Core i7
  - 3.8, 3.9, 3.12
- SMT: Exploiting Thread-Level Parallelism to Improve Uniprocessor Throughput
  - 3.11

Review:
Overcoming Data Hazards With Dynamic Scheduling
Textbook CAQA 3.4

Instruction Scheduling

I_1	FDIV.D	f6,	f6,	f4
I_2	FLD	f2,	45(x3)	
I_3	FMULT.D	f0,	f2,	f4
I_4	FDIV.D	f8,	f6,	f2
I_5	FSUB.D	f10,	f0,	f6
I_6	FADD.D	f6,	f8,	f2



Valid orderings:

in-order	I_1	I_2	I_3	I_4	I_5	I_6
out-of-order	I_2	I_1	I_3	I_4	I_5	I_6
out-of-order	I_1	I_2	I_3	I_5	I_4	I_6

Register Renaming for Eliminating WAR and WAW Dependencies

- Example:

```
DIV.D    F0,F2,F4
ADD.D    F6,F0,F8
S.D      F6,0(R1)
SUB.D    F8,F10,F14
MUL.D    F6,F10,F8
```

```
DIV.D    F0,F2,F4
ADD.D    F6,F0,F8
S.D      F6,0(R1)
SUB.D    T2,F10,F14
MUL.D    T1,F10,T2
```

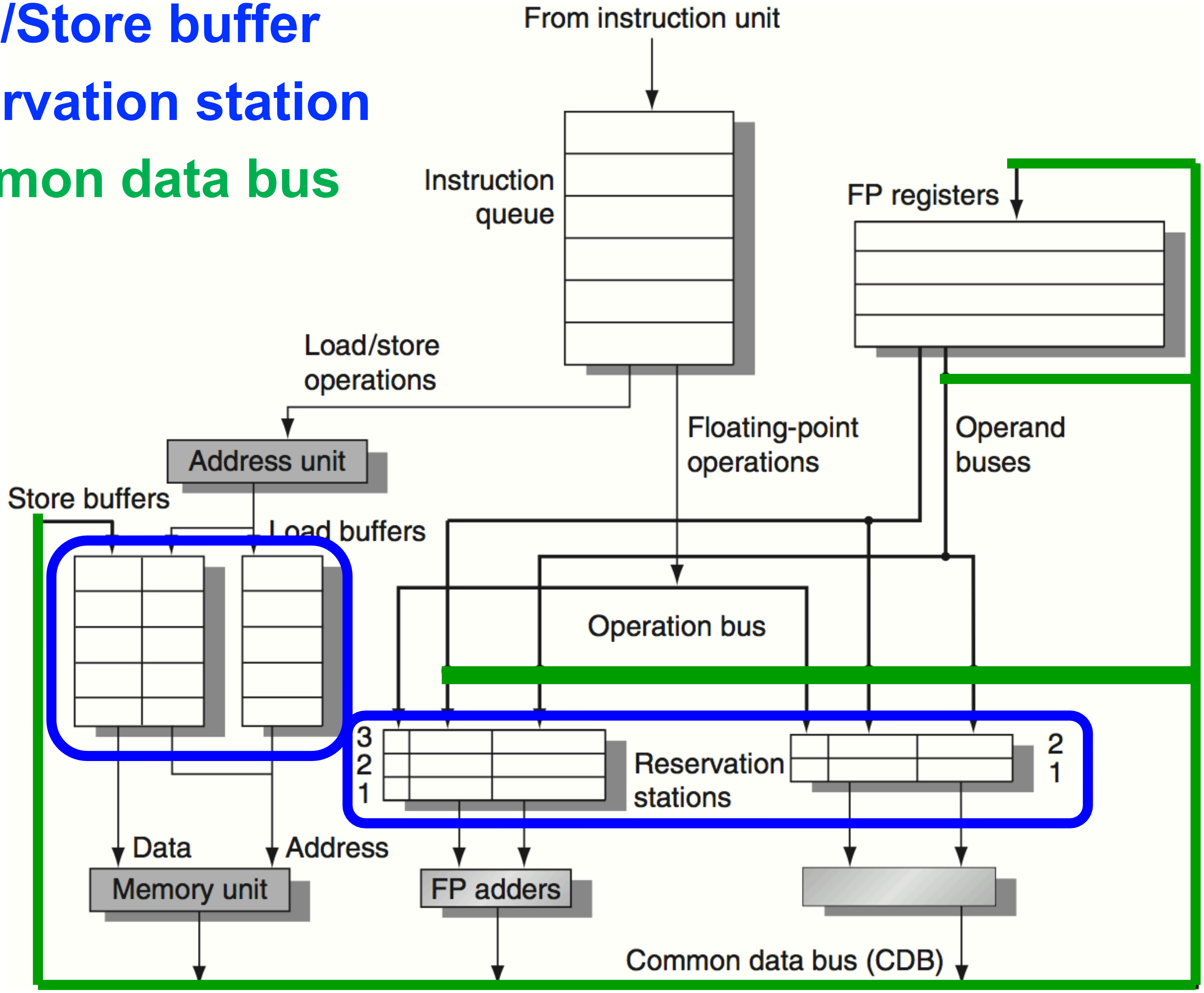
- Now only RAW hazards remain, which can be strictly ordered

Hardware Solution for Addressing Data Hazards

- Dynamic Scheduling of Instructions:
  - In-order issue
  - Out-of-order execution
  - Out-of-order completion
- Data Hazard via Register Renaming
  - Dynamic RAW hazard detection and scheduling in data-flow fashion
  - Register renaming for WRW and WRA hazard (name conflict)
- Implementations
  - Scoreboard (CDC 6600 1963)
    - » Centralized register renaming
  - Tomasulo's Approach (IBM 360/91, 1966)
    - » Distributed control and renaming via reservation station, load/store buffer and common data bus (data+source)

Organizations of Tomasulo's Algorithm

- Load/Store buffer
- Reservation station
- Common data bus



Three Stages of Tomasulo Algorithm

1. **Issue**—get instruction from FP Op Queue

   If reservation station free (no structural hazard),
   control issues instr & sends operands (renames registers).

2. **Execution**—operate on operands (EX)

   When both operands ready then execute;
   if not ready, watch Common Data Bus for result

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

   Write on Common Data Bus to all awaiting units;
   mark reservation station available

- Normal data bus: data + destination ("go to" bus)
- Common data bus: data + source ("come from" bus)
  - 64 bits of data + 4 bits of Functional Unit source address
  - Write if matches expected Functional Unit (produces result)
  - Does the broadcast

Tomasulo Example Cycle 3

Instruction status:

Instruction		j	k	Issue	Exec Comp	Write Result
LD	F6	34+	R2	1	3	
LD	F2	45+	R3	2		
MULTD	F0	F2	F4	3		
SUBD	F8	F6	F2			
DIVD	F10	F0	F6			
ADDD	F6	F8	F2			

	Busy	Address
Load1	Yes	34+R2
Load2	Yes	45+R3
Load3	No	

Reservation Stations:

Time	Name	Busy	Op	S1 Vj	S2 Vk	RS Qj	RS Qk
	Add1	No					
	Add2	No					
	Add3	No					
	Mult1	Yes	MULTD		R(F4)	Load2	
	Mult2	No					

Register result status:

Clock		F0	F2	F4	F6	F8	F10	F12	...	F30
3	*FU*	Mult1	Load2		Load1					

- **Note: registers names are removed ("renamed") in Reservation Stations**

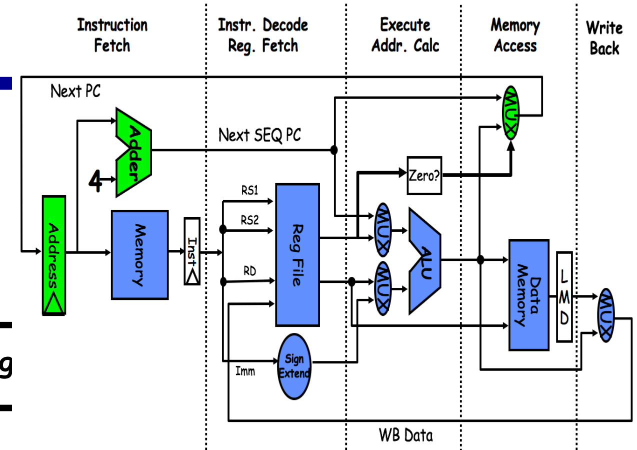
Register Renaming Summary

- **Purpose of Renaming: removing "Anti-dependencies"**
  - Get rid of WAR and WAW hazards, since these are not "real" dependencies
- **Implicit Renaming: i.e. Tomasulo**
  - Registers changed into values or response tags
  - We call this "implicit" because space in register file may or may not be used by results!
- **Explicit Renaming: more physical registers than needed by ISA.**
  - Rename table: tracks current association between architectural registers and physical registers
  - Uses a translation table to perform compiler-like transformation on the fly

Hardware-Based Speculation to Overcome Control Hazards

Textbook: CAQA 3.6

Control Hazard from Branches: Two or Three Cycles of Stall if Taken



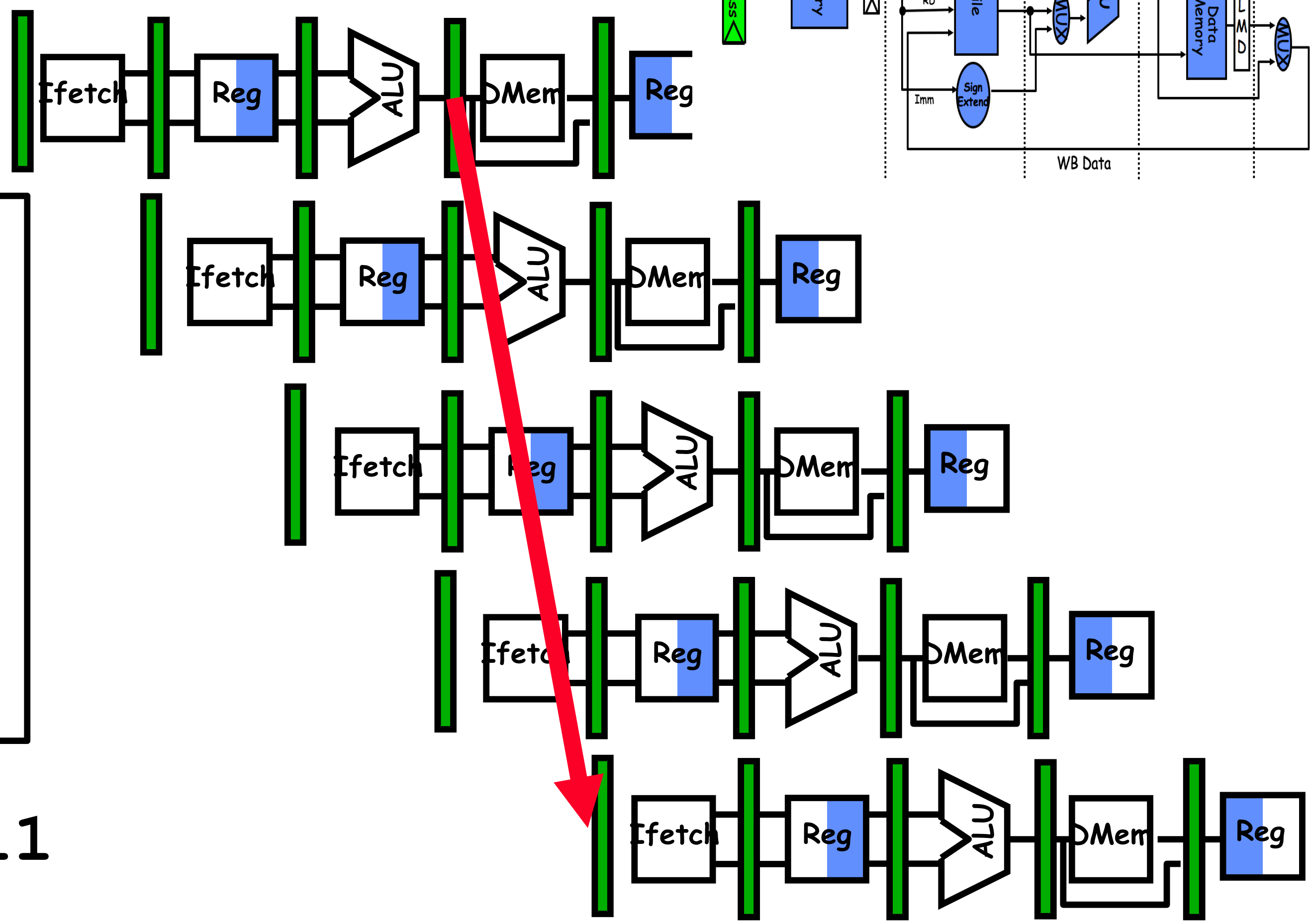
10: BEQ R1,R3,36

14: AND R2,R3,R5

18: OR R6,R1,R7

22: ADD R8,R1,R9

36: XOR R10,R1,R11

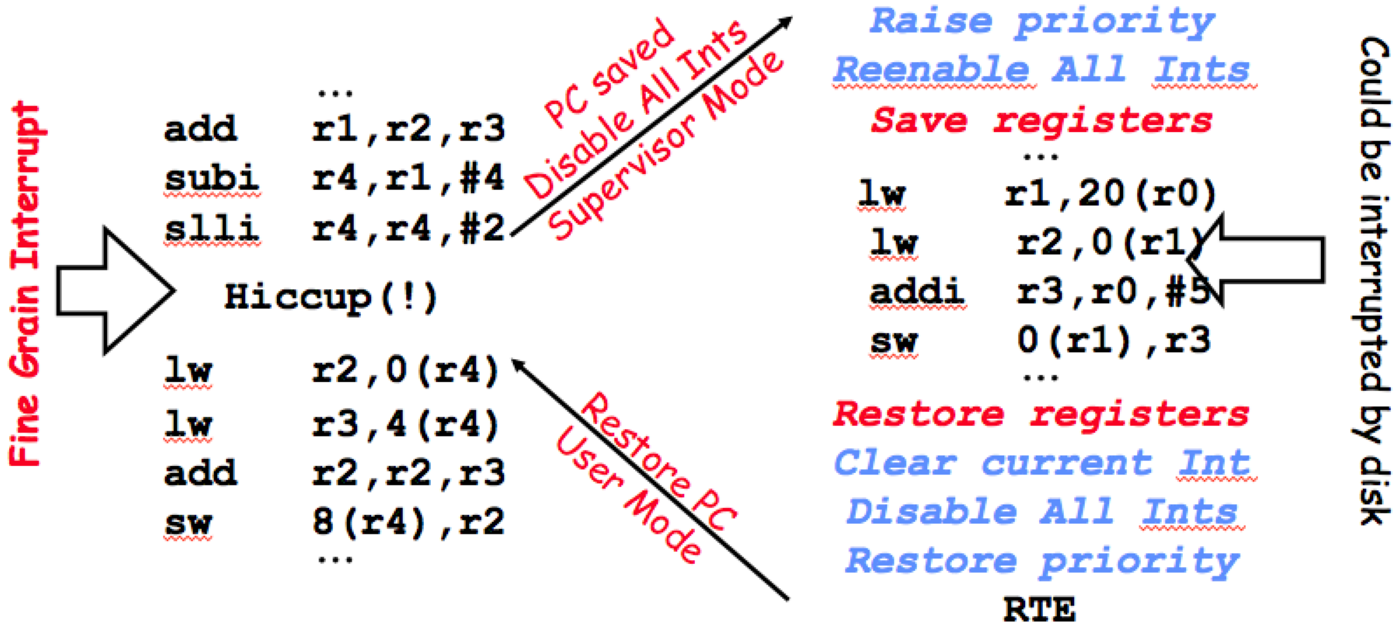


What do you do with the 3 instructions in between?

Control Hazards

- Break the instruction flow

- Unconditional Jump
- Conditional Jump
- Function call and return
- Exceptions



Branches Must Be Resolved Quickly

- The loop-unrolling example
  – we relied on the fact that branches were under control of "fast" integer unit in order to get overlap!

- Loop:

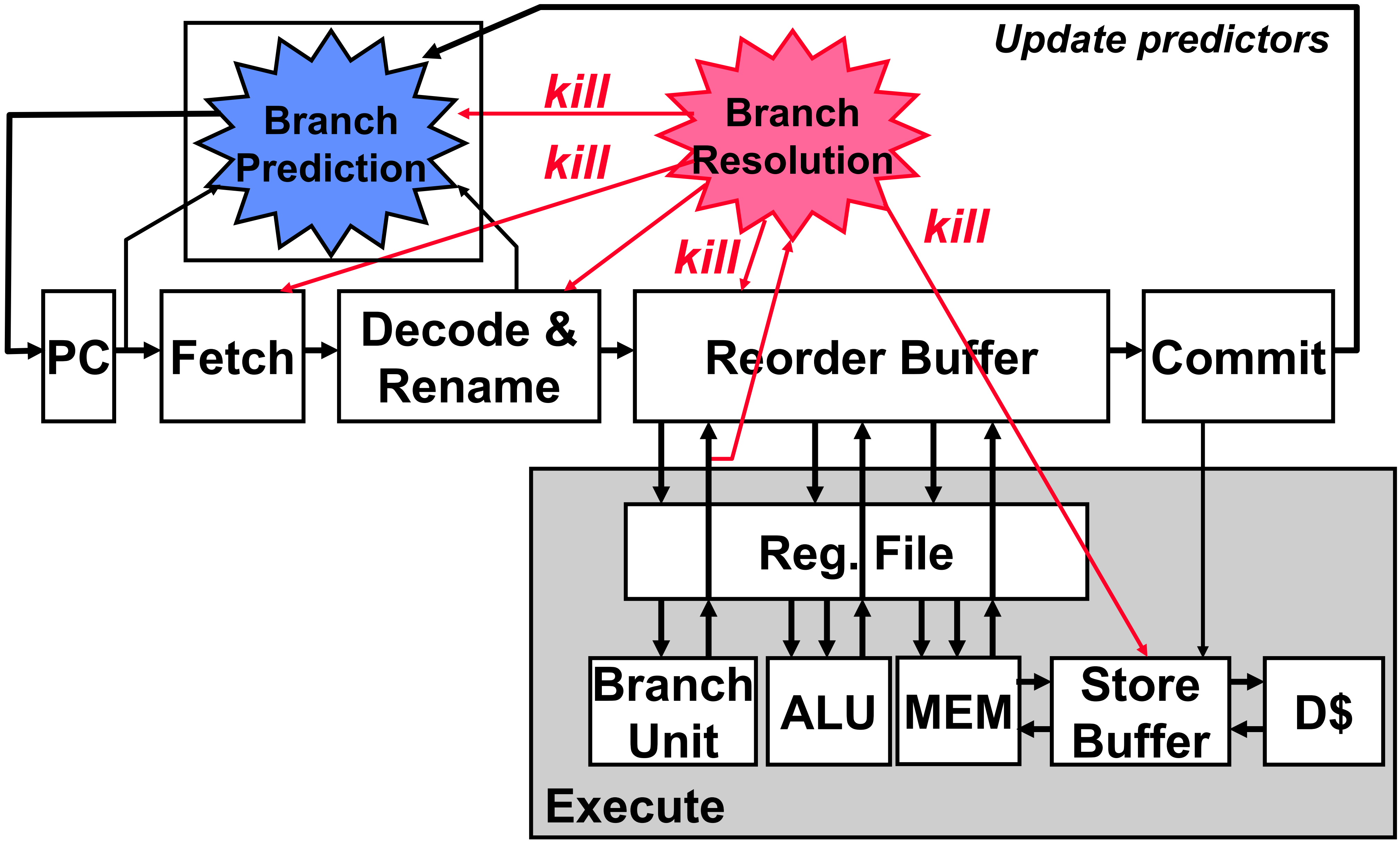
```
Loop:   LD      F0   0     R1
        MULTD   F4   F0    F2
        SD      F4   0     R1
        SUBI    R1   R1    #8
        BNEZ    R1   Loop
```

- What happens if branch depends on result of multd??
  – We completely lose all of our advantages!
  – Need to be able to "predict" branch outcome.
    » If we were to predict that branch was taken, this would be right most of the time.
- Problem much worse for superscalar (issue multiple instrs per cycle) machines!

Reducing Control Flow Penalty

- **Software solutions**
  - Eliminate branches - loop unrolling
    - » Increases the run length
  - Reduce resolution time - instruction scheduling
    - » Compute the branch condition as early as possible (of limited value)
- **Hardware solutions**
  - Find something else to do - delay slots
    - » Replaces pipeline bubbles with useful work (requires software cooperation)
- ***Branch speculation***
  - *Speculative (predicted) execution of instructions beyond the branch*
  - *Recover mis-predicted branch and its side-effect*

Speculation: Prediction + Mis-prediction Recovery



Branch Prediction

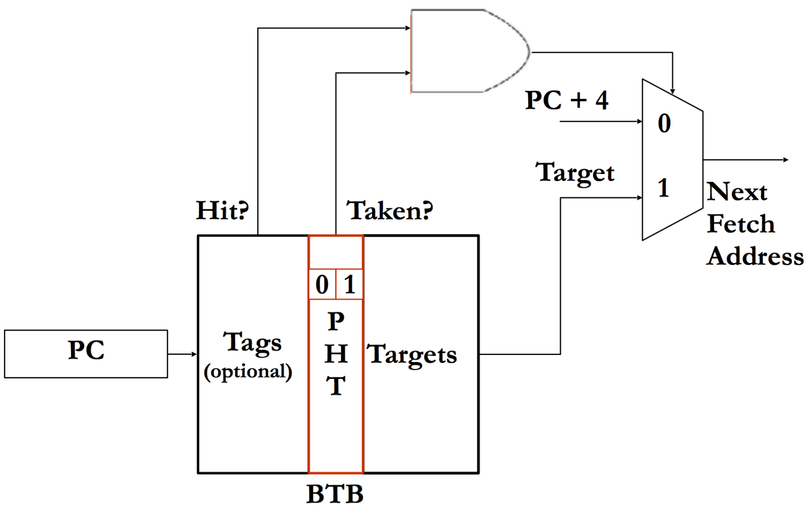
- **Motivation**
  - Branch penalties limit performance of deeply pipelined processors
- **Prediction works because Future can be predicted from past!**
  - Programs have patterns and hw just have to figure out what they are
  - Modern branch predictors have high accuracy: (>95%) and can reduce branch penalties significantly

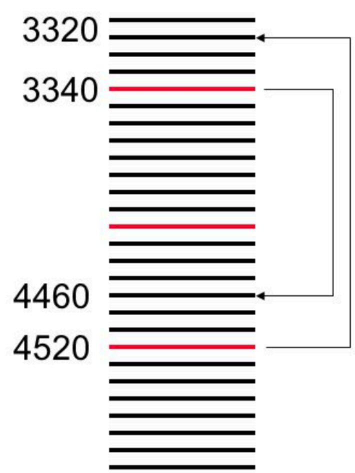
```
for (i=999; i>=0; i=i-1)
        x[i] = x[i] + s;
```

```
Loop:  fld      f0,0(x1)       //f0=array element
       fadd.d   f4,f0,f2       //add scalar in f2
       fsd      f4,0(x1)       //store result
       addi     x1,x1,-8       //decrement pointer
                               //8 bytes (per DW)
       bne      x1,x2,Loop     //branch x1≠x2
```

Branch Prediction

- Required hardware support
  - Branch history tables (Taken or Not)
  - Branch target buffers, etc. (Target address)





PC	Target PC	Prediction
3340	4460	1(T)
4520	3320	1(T)

```
Loop:  fld      f0,0(x1)       //f0=array element
       fadd.d   f4,f0,f2       //add scalar in f2
       fsd      f4,0(x1)       //store result
       addi     x1,x1,−8       //decrement pointer
                               //8 bytes (per DW)
       bne      x1,x2,Loop     //branch x1≠x2
```

Mispredict Recovery

In-order execution machines:

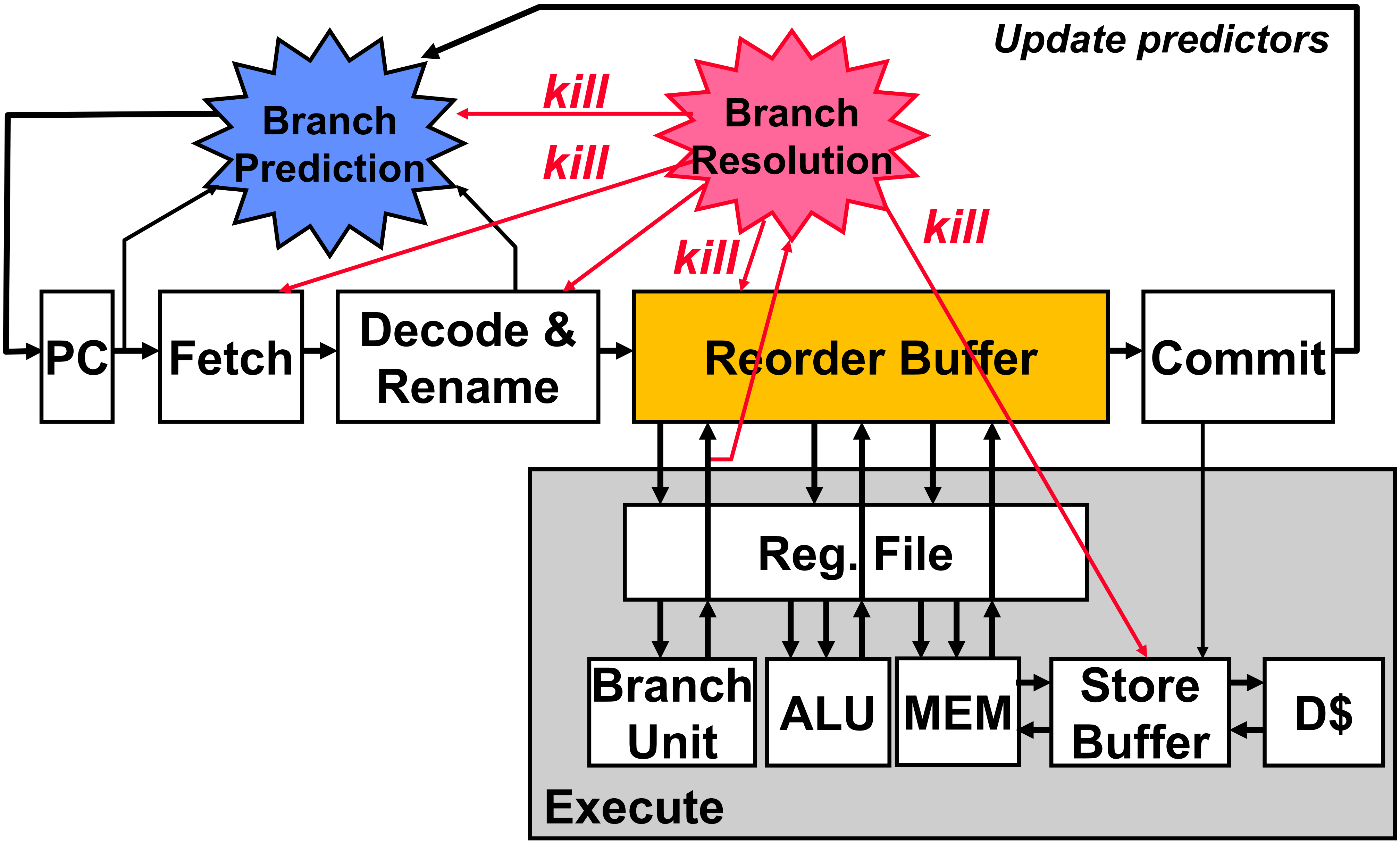
- Assume no instruction issued after branch can write-back before branch resolves
- Kill all instructions in pipeline behind mispredicted branch

Out-of-order execution:

- Multiple instructions following branch in program order can complete before branch resolves

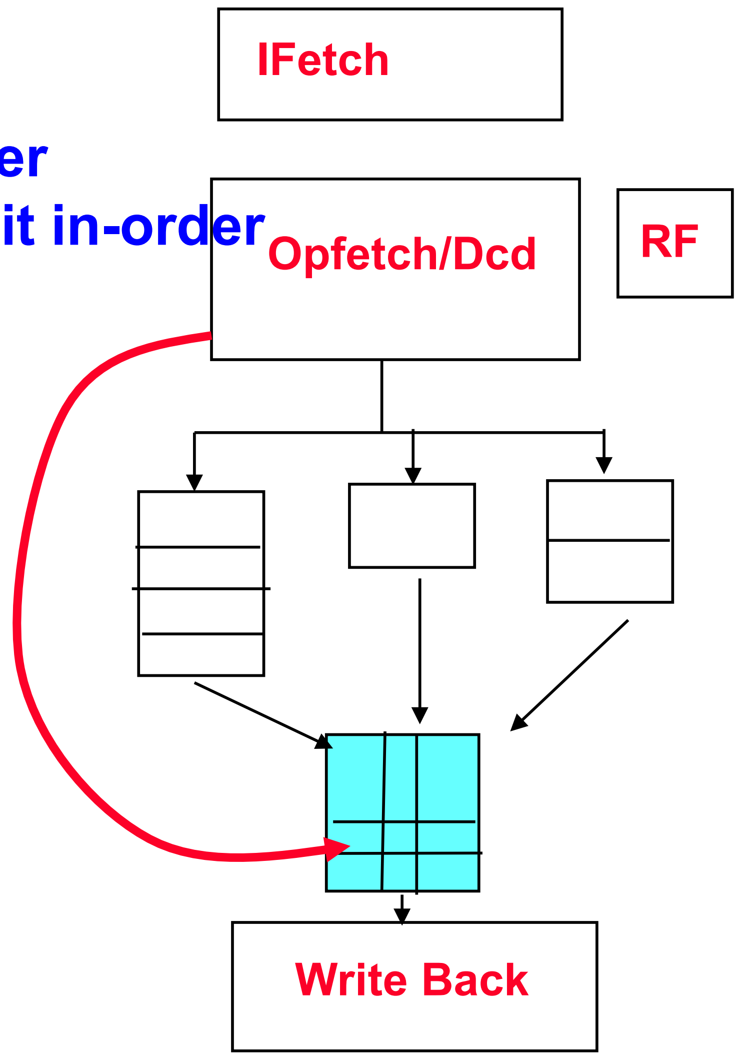
- Temporary store the intermediate state for those instructions that may be cancelled
  - Keep result computation separate from commit
  - Kill instructions following branch in pipeline
  - Restore state to state following branch

Branch Prediction/Speculation



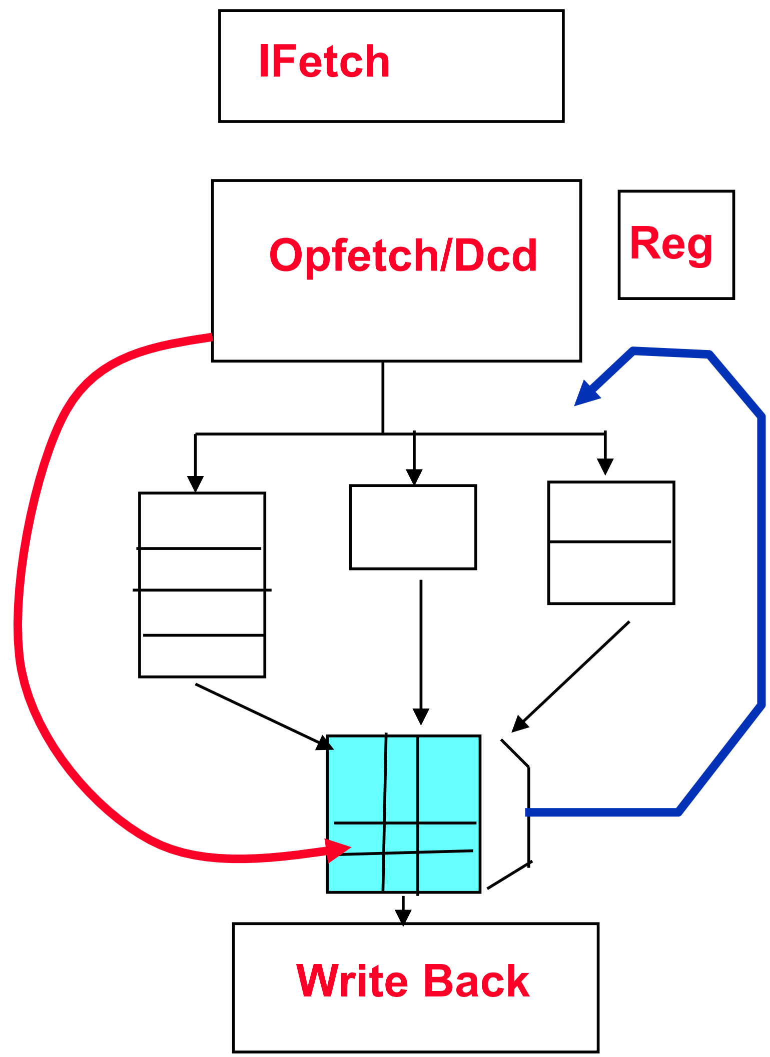
Reorder Buffer is a FIFO Queue

- **Idea:**
  - Record instruction issue order
  - Allow them to execute out of order
  - Reorder them so that they commit in-order
- **On issue:**
  - Reserve slot at tail of ROB
  - Record dest reg, PC
  - Tag u-op with ROB slot
- **Done execute**
  - Deposit result in ROB slot
  - Mark exception state
- **WB head of ROB**
  - Check exception, handle
  - Write register value, or
  - Commit the store



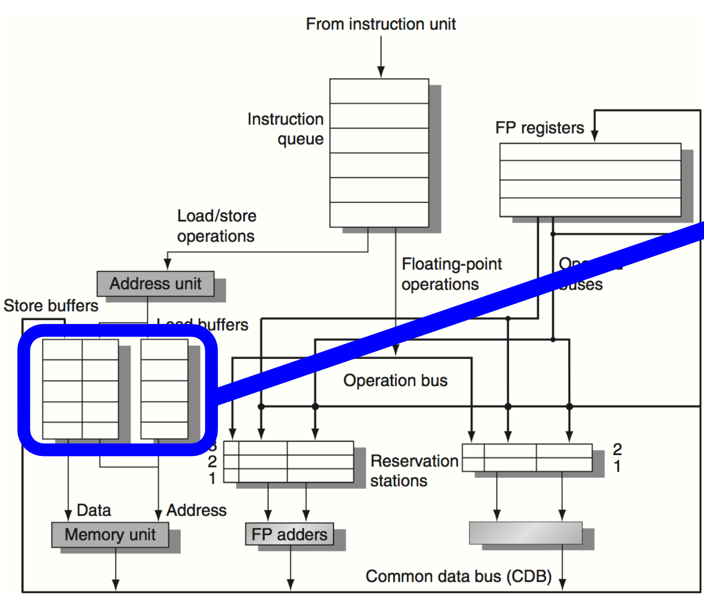
Reorder Buffer + Forwarding + Speculation

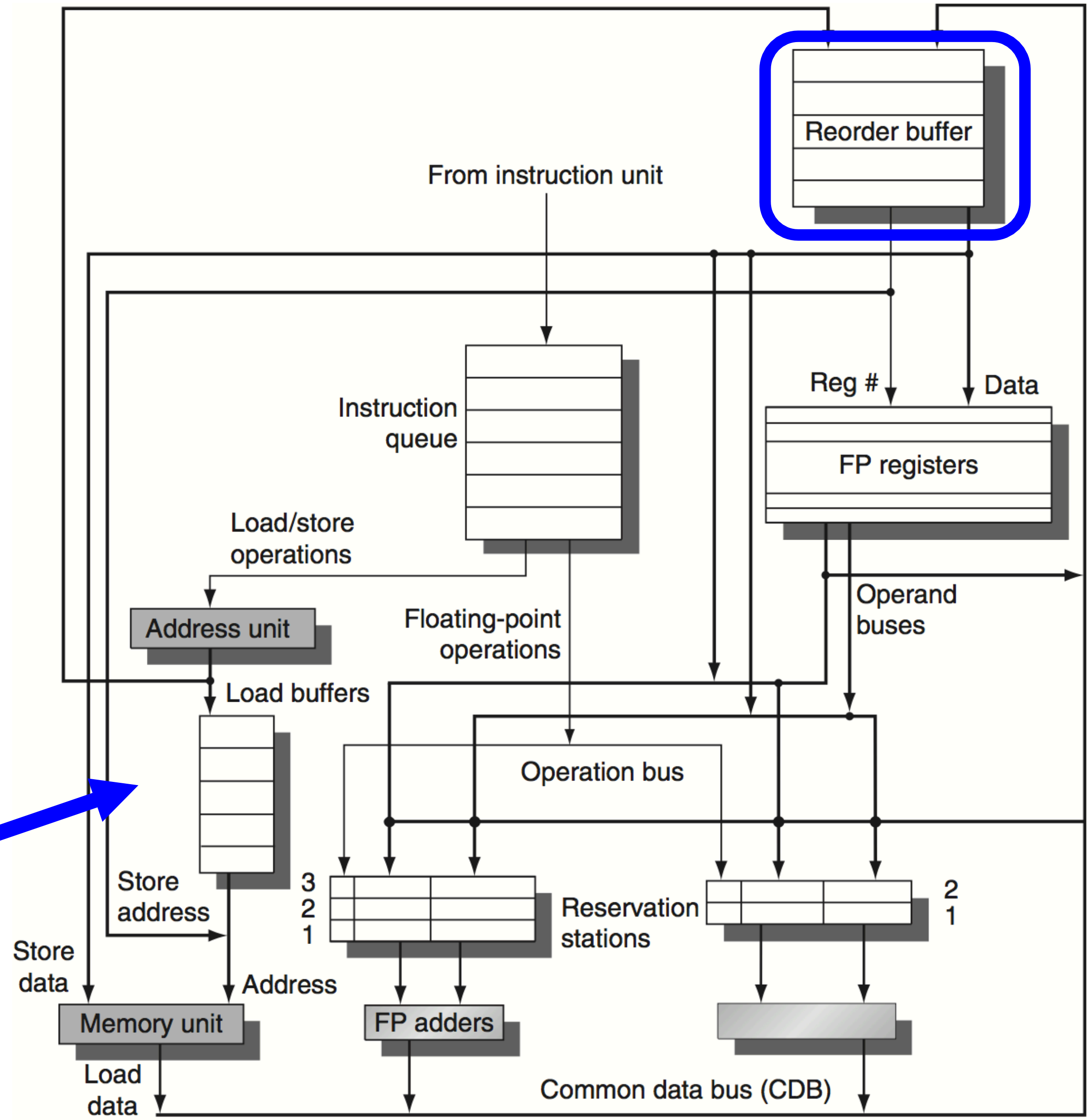
- Idea:
  - Issue branch into ROB
  - Mark with prediction
  - Fetch and issue predicted instructions speculatively
  - Branch must resolve before leaving ROB
  - Resolve correct
    - » Commit following instr
  - Resolve incorrect
    - » Mark following instr in ROB as invalid
    - » Let them clear



Hardware Speculation in Tomasulo Algorithm

- **+ Reorder Buffer**
- **- Store Buffer**
  - **Integrated in ROF**





Four Steps of *Speculative* Tomasulo

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")

Instruction In-order Commit

- **Also called completion or graduation**
- **In-order commit**
  - **In-order issue**
  - **Out-of-order execution**
  - **Out-of-order completion**
- **Three cases when an instr reaches the head of ROB**
  - **Normal commit: when an instruction reaches the head of the ROB and its result is present in the buffer**
    - » **The processor updates the register with the result and removes the instruction from the ROB.**
  - **Committing a store:**
    - » **is similar except that memory is updated rather than a result register.**
  - **A branch with incorrect prediction**
    - » **indicates that the speculation was wrong.**
    - » **The ROB is flushed and execution is restarted at the correct successor of the branch.**

Example with ROB and Reservation (Dynamic Scheduling and Speculation)

- **MUL.D is ready to commit**

Reorder buffer

Entry	Busy	Instruction		State	Destination	Value
1	No	L.D	F6,32(R2)	Commit	F6	Mem[32 + Regs[R2]]
2	No	L.D	F2,44(R3)	Commit	F2	Mem[44 + Regs[R3]]
3	Yes	MUL.D	F0,F2,F4	Write result	F0	#2 × Regs[F4]
4	Yes	SUB.D	F8,F2,F6	Write result	F8	#2 – #1
5	Yes	DIV.D	F10,F0,F6	Execute	F10	
6	Yes	ADD.D	F6,F8,F2	Write result	F6	#4 + #2

Reservation stations

Name	Busy	Op	Vj	Vk	Qj	Qk	Dest	A
Load1	No							
Load2	No							
Add1	No							
Add2	No							
Add3	No							
Mult1	No	MUL.D	Mem[44 + Regs[R3]]	Regs[F4]			#3	
Mult2	Yes	DIV.D		Mem[32 + Regs[R2]]	#3		#5	

After SUB.D completes execution, if exception happens by MUL.D ….

FP register status

Field	F0	F1	F2	F3	F4	F5	F6	F7	F8	F10
Reorder #	3						6		4	5
Busy	Yes	No	No	No	No	No	Yes	. . .	Yes	Yes

Figure 3.12 At the time the MUL.D is ready to commit, only the two L.D instructions have committed, although

In-order ~~Commit with~~ Branch

```
Loop:   L.D      F0,0(R1)
        MUL.D    F4,F0,F2
        S.D      F4,0(R1)
        DADDIU   R1,R1,#-8
        BNE      R1,R2,Loop     ;branches if R1|
```

Reorder buffer

Entry	Busy	Instruction	State	Destination	Value
1	No	L.D F0,0(R1)	Commit	F0	Mem[0 + Regs[R1]]
2	No	MUL.D F4,F0,F2	Commit	F4	#1 × Regs[F2]
3	Yes	S.D F4,0(R1)	Write result	0 + Regs[R1]	#2
4	Yes	DADDIU R1,R1,#-8	Write result	R1	Regs[R1] – 8
5	Yes	BNE R1,R2,Loop	Write result		
6	Yes	L.D F0,0(R1)	Write result	F0	Mem[#4]
7	Yes	MUL.D F4,F0,F2	Write result	F4	#6 × Regs[F2]
8	Yes	S.D F4,0(R1)	Write result	0 + #4	#7
9	Yes	DADDIU R1,R1,#-8	Write result	R1	#4 – 8
10	Yes	BNE R1,R2,Loop	Write result		

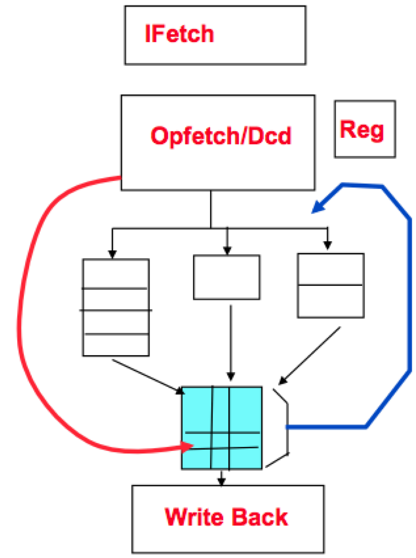
FLUSHED

IF Misprediction

FP register status

Field	F0	F1	F2	F3	F4	F5	F6	F7	F8
Reorder #	6				7				
Busy	Yes	No	No	No	Yes	No	No	...	No

Summary: Dynamic Scheduling and Speculation



- **ILP Maximized (a restricted data-flow)**
  - In-order issue
  - Out-of-order execution
  - Out-of-order completion
  - In-order commit
- **Data Hazards**
  - Input operands-driven dynamic scheduling for RAW hazard
  - Register renaming for handling WAR and WAW hazards
- **Control Hazards (Branching, Precision Exception)**
  - Branch prediction and in-order commit
- **Implementation: Tomasulo**
  - Reservation stations and Reorder buffer
  - Other solutions as well (scoreboard, history table)

Multiple ISSUE via VLIW/Static Superscalar

Textbook: CAQA 3.7

Multiple Issue

IF	ID	EX	MEM	WB			
IF	ID	EX	MEM	WB			
	IF	ID	EX	MEM	WB		
	IF	ID	EX	MEM	WB		
		IF	ID	EX	MEM	WB	
		IF	ID	EX	MEM	WB	
			IF	ID	EX	MEM	W
			IF	ID	EX	MEM	W
				IF	ID	EX	ME
				IF	ID	EX	ME

- Issue multiple instructions in one cycle
- Three major types (VLIW and superscalar)
  - Statically scheduled superscalar processors
  - VLIW (very long instruction word) processors
  - Dynamically scheduled superscalar processors
- Superscalar
  - Variable # of instr per cycle
  - In-order execution for static superscalar
  - Out-of-order execution for dynamic superscalar
- VLIW
  - Issue a fixed number of instructions formatted either as one large instruction or as a fixed instruction packet with the parallel- ism among instructions explicitly indicated by the instruction.
  - Inherently statically scheduled by the compiler
  - Intel/HP IA-64 architecture, named EPIC—explicitly parallel instruction computer
    - » Appendix H,

Comparison

Common name	Issue structure	Hazard detection	Scheduling	Distinguishing characteristic	Examples
Superscalar (static)	Dynamic	Hardware	Static	In-order execution	Mostly in the embedded space: MIPS and ARM, including the ARM Cortex-A8
Superscalar (dynamic)	Dynamic	Hardware	Dynamic	Some out-of-order execution, but no speculation	None at the present
Superscalar (speculative)	Dynamic	Hardware	Dynamic with speculation	Out-of-order execution with speculation	Intel Core i3, i5, i7; AMD Phenom; IBM Power 7
VLIW/LIW	Static	Primarily software	Static	All hazards determined and indicated by compiler (often implicitly)	Most examples are in signal processing, such as the TI C6x
EPIC	Primarily static	Primarily software	Mostly static	All hazards determined and indicated explicitly by the compiler	Itanium

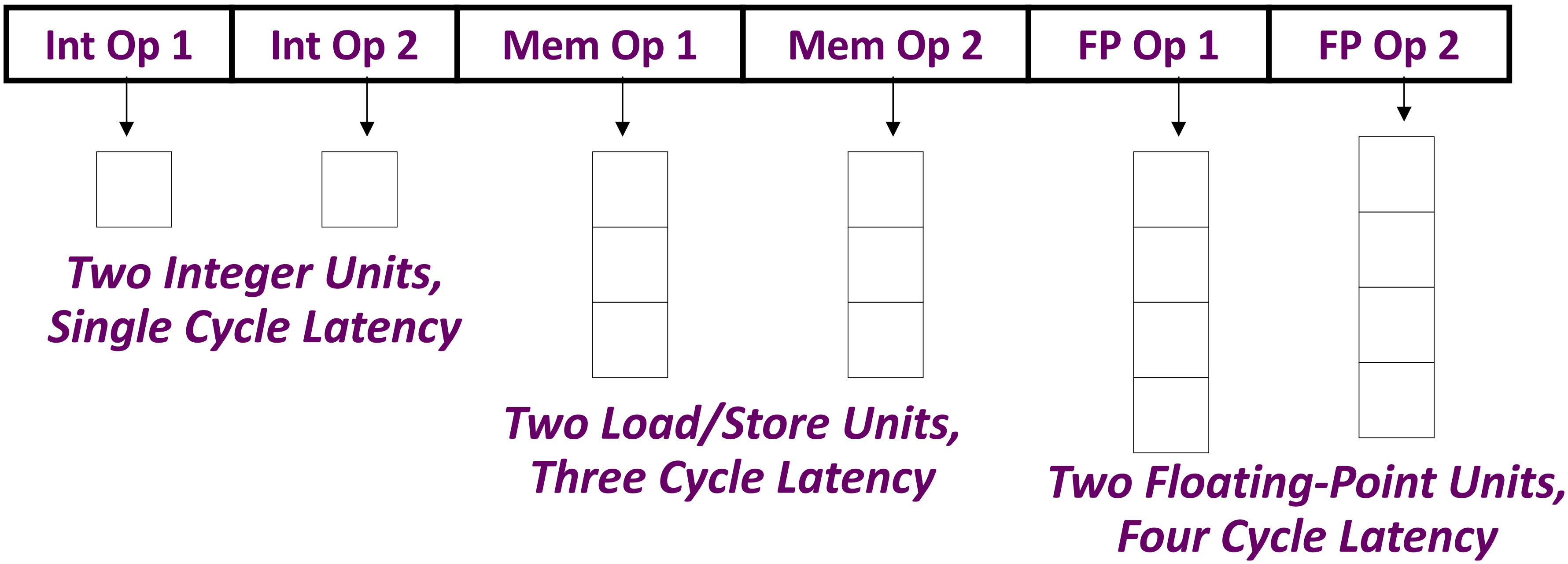
Figure 3.15 The five primary approaches in use for multiple-issue processors and the primary characteristics that distinguish them. This chapter has focused on the hardware-intensive techniques, which are all some form of superscalar. Appendix H focuses on compiler-based approaches. The EPIC approach, as embodied in the IA-64 architecture, extends many of the concepts of the early VLIW approaches, providing a blend of static and dynamic approaches.

VLIW and Static Superscalar

- **Very similar in terms of the requirements for compiler and hardware support**
- **We will discuss VLIW**

- **Very Long Instruction Word (VLIW)**
  - **packages the multiple operations into one very long instruction**

VLIW: Very Long Instruction Word



- **Multiple operations packed into one instruction**
- **Each operation slot is for a fixed function**
- **Constant operation latencies are specified**
- **Architecture requires guarantee of:**
  - **Parallelism within an instruction => no cross-operation RAW check**
  - **No data use before data ready => no data interlocks**

VLIW: Very Large Instruction Word

- Each "instruction" has explicit coding for multiple operations
  - In IA-64, grouping called a "packet"
  - In Transmeta, grouping called a "molecule" (with "atoms" as ops)
- Tradeoff instruction space for simple decoding
  - The long instruction word has room for many operations
  - By definition, all the operations the compiler puts in the long instruction word are independent => execute in parallel
  - E.g., 1 integer operation/branch, 2 FP ops, 2 Memory refs
    - » 16 to 24 bits per field => 5*16 or 80 bits to 5*24 or 120 bits wide
  - Need compiling technique that schedules across several branches

Recall: Unrolled Loop that Minimizes Stalls for Scalar

```
1  Loop: L.D     F0,0(R1)
2        L.D     F6,-8(R1)
3        L.D     F10,-16(R1)
4        L.D     F14,-24(R1)
5        ADD.D   F4,F0,F2
6        ADD.D   F8,F6,F2
7        ADD.D   F12,F10,F2
8        ADD.D   F16,F14,F2
9        S.D     0(R1),F4
10       S.D     -8(R1),F8
11       S.D     -16(R1),F12
12       DSUBUI  R1,R1,#32
13       BNEZ    R1,LOOP
14       S.D     8(R1),F16     ; 8-32 = -24
```

L.D to ADD.D: 1 Cycle
ADD.D to S.D: 2 Cycles

```
for (i=999; i>=0; i=i-1)
      x[i] = x[i] + s;
```

14 clock cycles, or 3.5 per iteration

Loop Unrolling in VLIW

Unrolled 7 times to avoid delays

7 results in 9 clocks, or 1.3 clocks per iteration (1.8X)

Average: 2.5 ops per clock, 50% efficiency

Memory reference 1	Memory reference 2	FP operation 1	FP operation 2	Integer operation/branch
L.D F0,0(R1)	L.D F6,-8(R1)			
L.D F10,-16(R1)	L.D F14,-24(R1)			
L.D F18,-32(R1)	L.D F22,-40(R1)	ADD.D F4,F0,F2	ADD.D F8,F6,F2	
L.D F26,-48(R1)		ADD.D F12,F10,F2	ADD.D F16,F14,F2	
		ADD.D F20,F18,F2	ADD.D F24,F22,F2	
S.D F4,0(R1)	S.D F8,-8(R1)	ADD.D F28,F26,F2		
S.D F12,-16(R1)	S.D F16,-24(R1)			DADDUI R1,R1,#-56
S.D F20,24(R1)	S.D F24,16(R1)			
S.D F28,8(R1)				BNE R1,R2,Loop

Figure 3.16 VLIW instructions that cycles assuming no branch delay; norr ations in 9 clock cycles, or 2.5 operati operation, is about 60%. To achieve thi this loop. The VLIW code sequence abc MIPS processor can use as few as two l

Instruction producing result	Instruction using result	Latency in clock cycles
FP ALU op	Another FP ALU op	3
FP ALU op	Store double	2
Load double	FP ALU op	1
Load double	Store double	0

Loop Unrolling in VLIW

- **Unroll 8 times**
  - **Enough registers**

8 results in 9 clocks, or 1.125 clocks per iteration

Average: 2.89 (26/9) ops per clock, 58% efficiency (26/45)

Memory reference 1	Memory reference 2	FP operation 1	FP operation 2	Integer operation/branch
L.D F0,0(R1)	L.D F6,-8(R1)			
L.D F10,-16(R1)	L.D F14,-24(R1)			
L.D F18,-32(R1)	L.D F22,-40(R1)	ADD.D F4,F0,F2	ADD.D F8,F6,F2	
L.D F26,-48(R1)	**L.D**	ADD.D F12,F10,F2	ADD.D F16,F14,F2	
		ADD.D F20,F18,F2	ADD.D F24,F22,F2	
S.D F4,0(R1)	S.D F8,-8(R1)	ADD.D F28,F26,F2	**ADD.D**	
S.D F12,-16(R1)	S.D F16,-24(R1)			DADDUI R1,R1,#-56
S.D F20,24(R1)	S.D F24,16(R1)			
S.D F28,8(R1)	**S.D**			BNE R1,R2,Loop

Figure 3.16 VLIW instructions that cycles assuming no branch delay; norr ations in 9 clock cycles, or 2.5 operati operation, is about 60%. To achieve thi this loop. The VLIW code sequence abc MIPS processor can use as few as two l

Instruction producing result	Instruction using result	Latency in clock cycles
FP ALU op	Another FP ALU op	3
FP ALU op	Store double	2
Load double	FP ALU op	1
Load double	Store double	0

Loop Unrolling in VLIW

Instruction producing result	Instruction using result	Latency in clock cycles
FP ALU op	Another FP ALU op	3
FP ALU op	Store double	2
Load double	FP ALU op	1
Load double	Store double	0

- Unroll 10 times
  - Enough registers

10 results in 10 clocks, or 1 clock per iteration

Average: 3.2 ops per clock (32/10), 64% efficiency (32/50)

Memory reference 1	Memory reference 2	FP operation 1	FP operation 2	Integer operation/branch
L.D F0,0(R1)	L.D F6,-8(R1)			
L.D F10,-16(R1)	L.D F14,-24(R1)			
L.D F18,-32(R1)	L.D F22,-40(R1)	ADD.D F4,F0,F2	ADD.D F8,F6,F2	
L.D F26,-48(R1)	L.D	ADD.D F12,F10,F2	ADD.D F16,F14,F2	
L.D	L.D	ADD.D F20,F18,F2	ADD.D F24,F22,F2	
S.D F4,0(R1)	S.D F8,-8(R1)	ADD.D F28,F26,F2	ADD.D	
S.D F12,-16(R1)	S.D F16,-24(R1)	ADD.D	ADD.D	
S.D F20,24(R1)	S.D F24,16(R1)			DADDUI R1,R1,#-56
S.D F28,8(R1)	S.D			
S.D	S.D			BNE R1,R2,Loop

Problems with 1st Generation VLIW

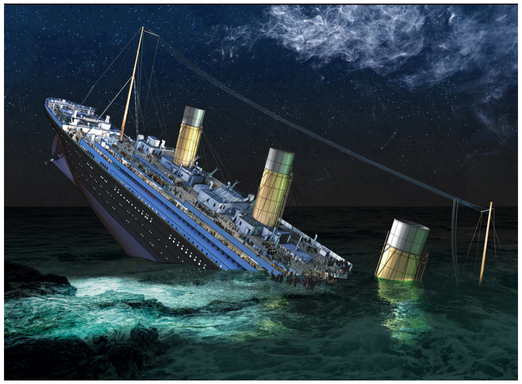
- **Increase in code size**
  - generating enough operations in a straight-line code fragment requires ambitiously unrolling loops
  - whenever VLIW instructions are not full, unused functional units translate to wasted bits in instruction encoding
- **Operated in lock-step; no hazard detection HW**
  - a stall in any functional unit pipeline caused entire processor to stall, since all functional units must be kept synchronized
  - Compiler might prediction function units, but caches hard to predict
- **Binary code compatibility**
  - Pure VLIW => different numbers of functional units and unit latencies require different versions of the code

Intel/HP IA-64 “Explicitly Parallel Instruction Computer (EPIC)”

- IA-64: instruction set architecture
  - 128 64-bit integer regs + 128 82-bit floating point regs
    - » Not separate register files per functional unit as in old VLIW
  - Hardware checks dependencies (interlocks ⇒ binary compatibility over time)
- 3 Instructions in 128 bit “bundles”; field determines if instructions dependent or independent
  - Smaller code size than old VLIW, larger than x86/RISC
  - Groups can be linked to show independence > 3 instr
- Predicated execution (select 1 out of 64 1-bit flags) ⇒ 40% fewer mispredictions?
- Speculation Support:
  - deferred exception handling with “poison bits”
  - Speculative movement of loads above stores + check to see if incorect
- Itanium™ was first implementation (2001)
  - Highly parallel and deeply pipelined hardware at 800Mhz
  - 6-wide, 10-stage pipeline at 800Mhz on 0.18 µ process
- Itanium 2™ is name of 2nd implementation (2005)
  - 6-wide, 8-stage pipeline at 1666Mhz on 0.13 µ process
  - Caches: 32 KB I, 32 KB D, 128 KB L2I, 128 KB L2D, 9216 KB L3

Summary

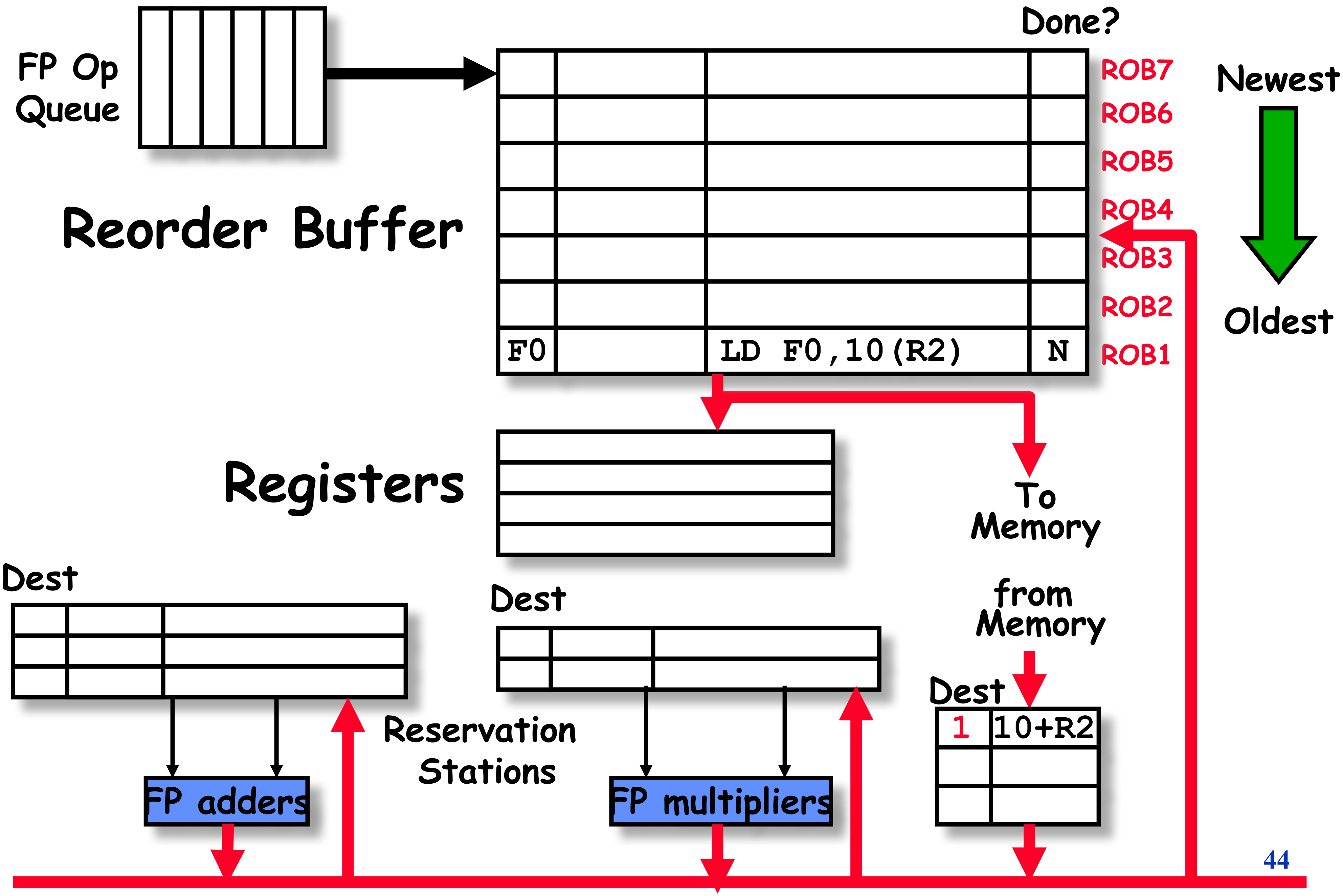
- VLIW: Explicitly Parallel, Static Superscalar
  – Requires advanced and aggressive compiler techniques
  – Trace Scheduling: Select primary "trace" to compress + fixup code
- Other aggressive techniques
  – Boosting: Moving of instructions above branches
    » Need to make sure that you get same result (i.e. do not violate dependencies)
    » Need to make sure that exception model is same (i.e. not unsafe)
- Itanium/EPIC/VLIW is not a breakthrough in ILP
  – If anything, it is as complex or more so than a dynamic processor
  – **Some refers to as Itanic!**
- BUT it is used today:
  – e.g. TI sigal processor C6x



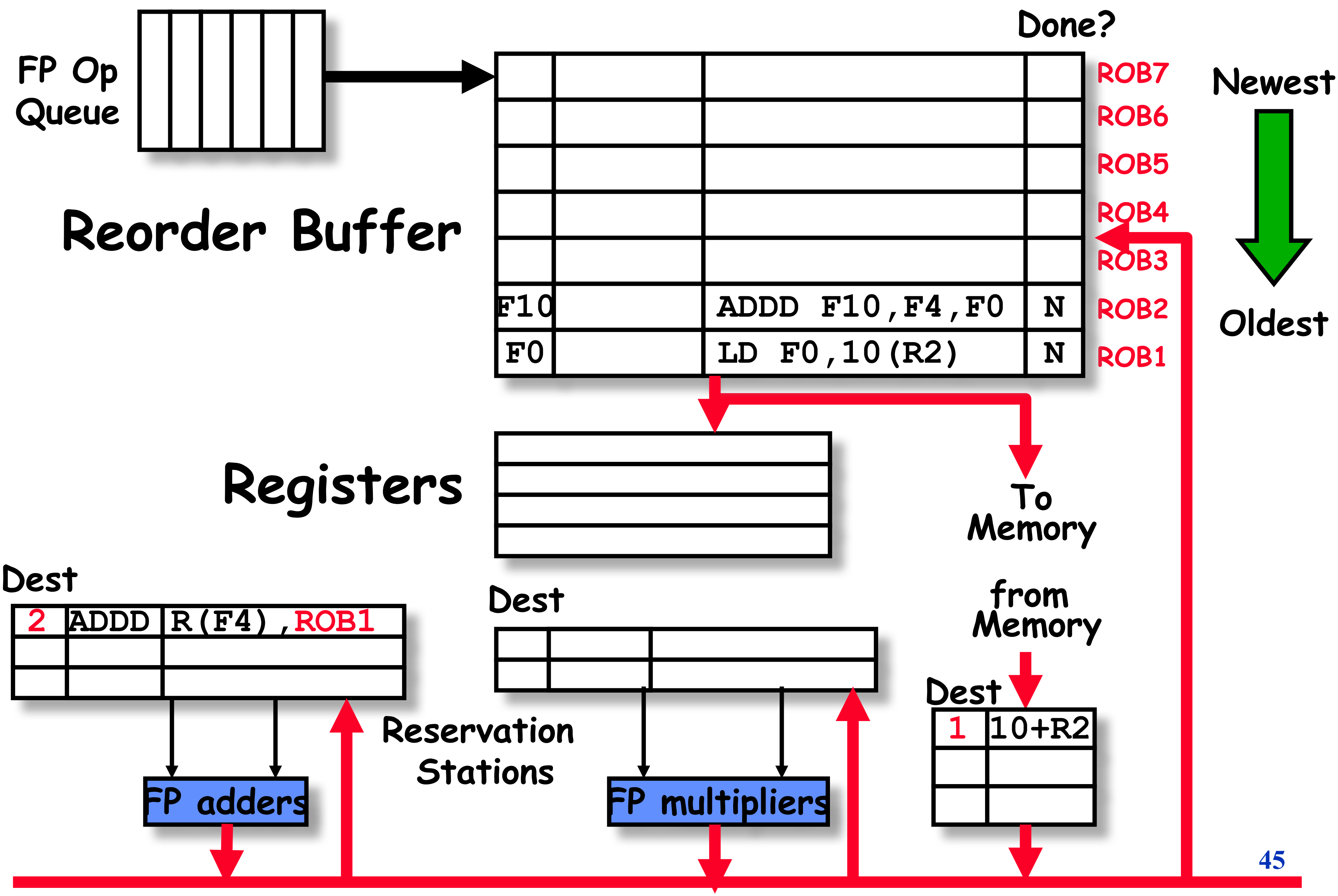
Class Lectures End Here!

SPECULATION EXAMPLE

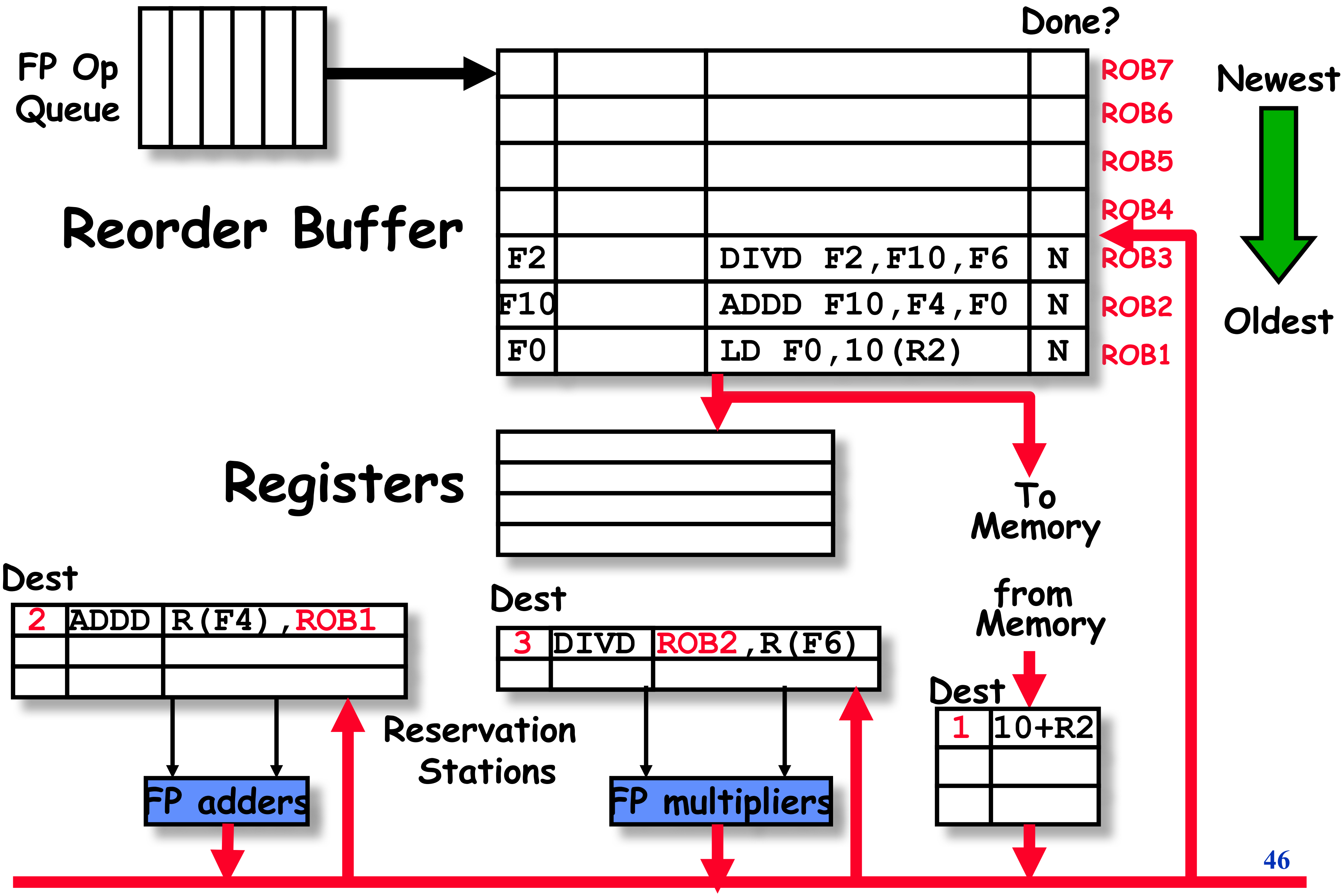
Tomasulo With Reorder buffer:



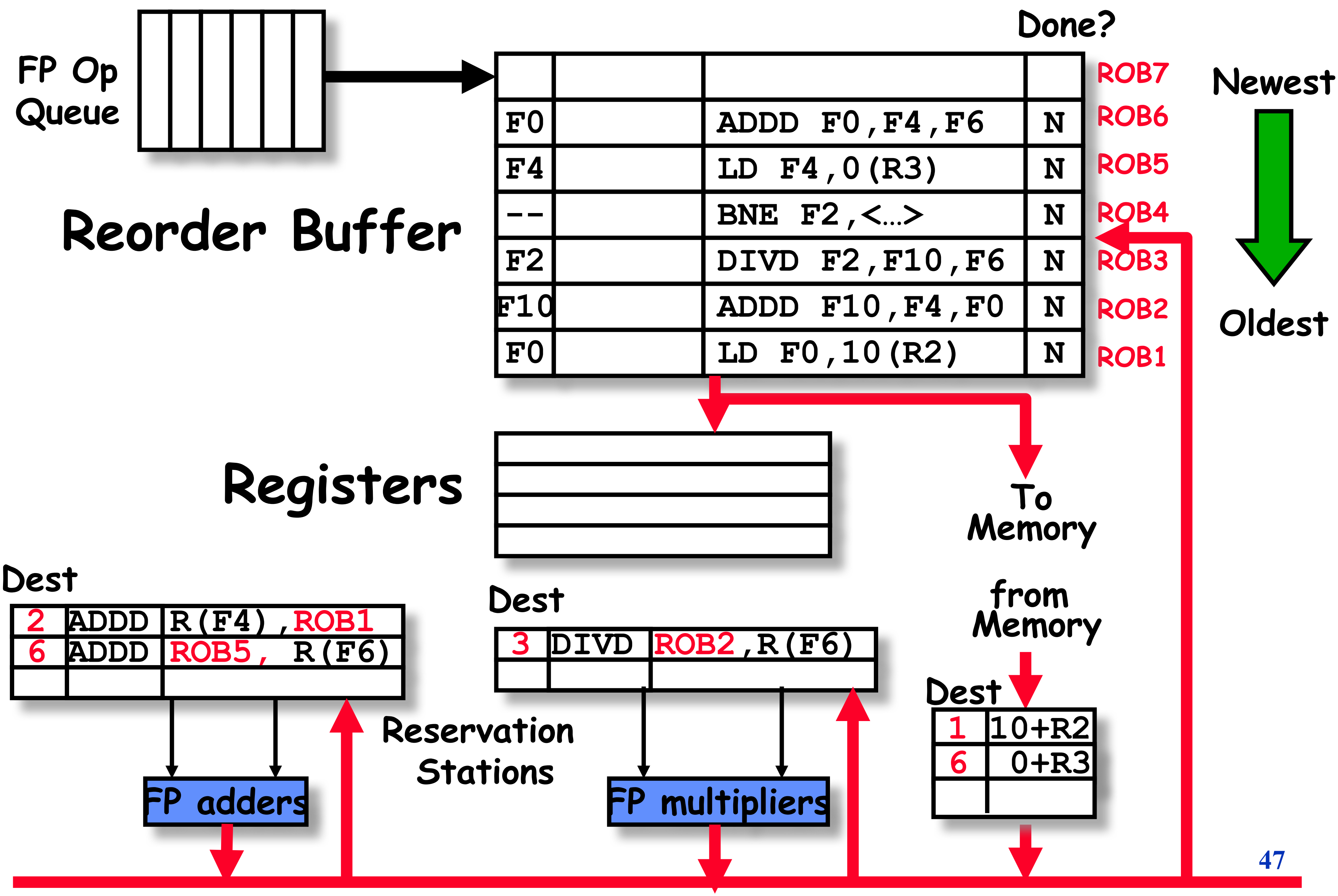
Tomasulo With Reorder buffer:



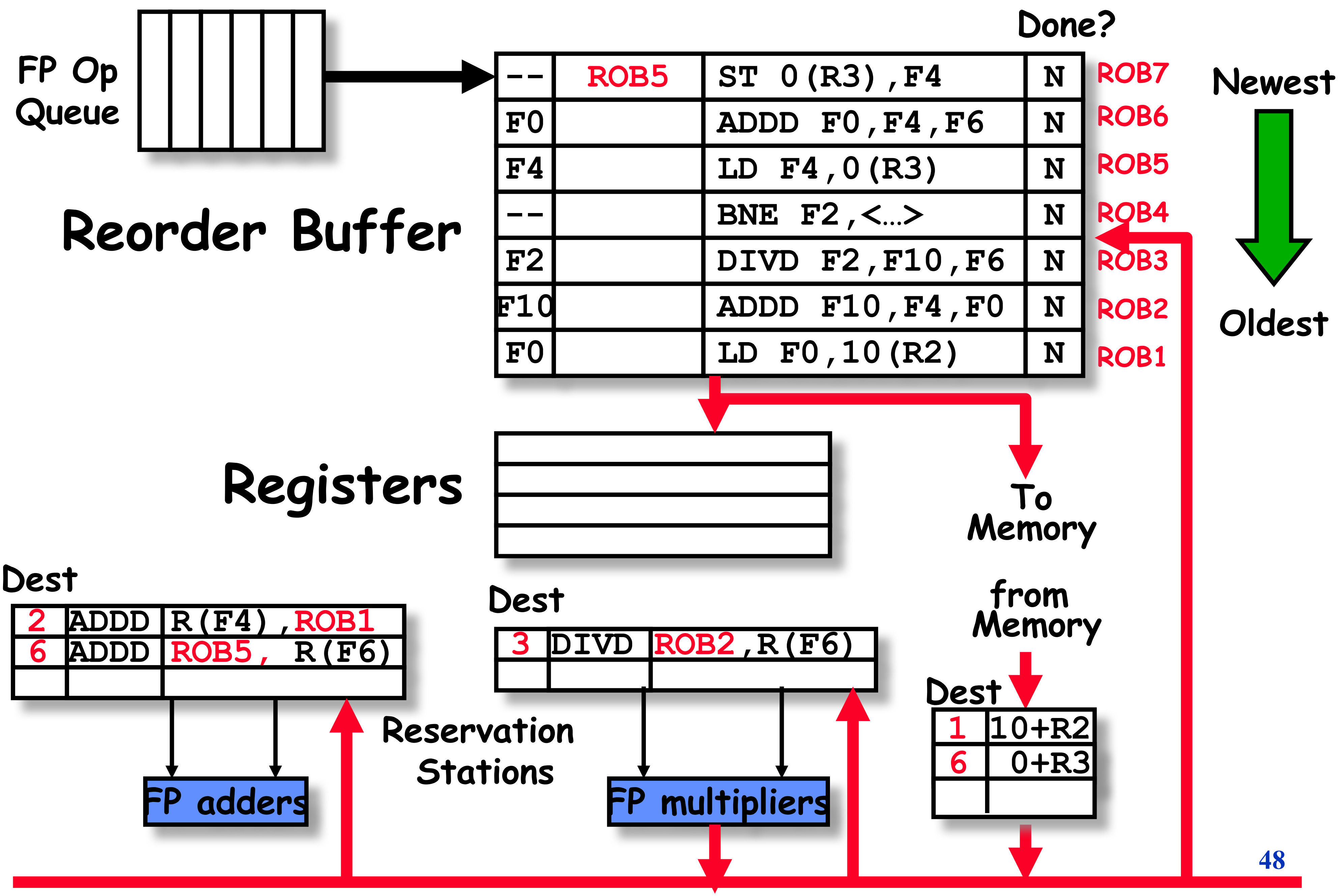
Tomasulo With Reorder buffer:



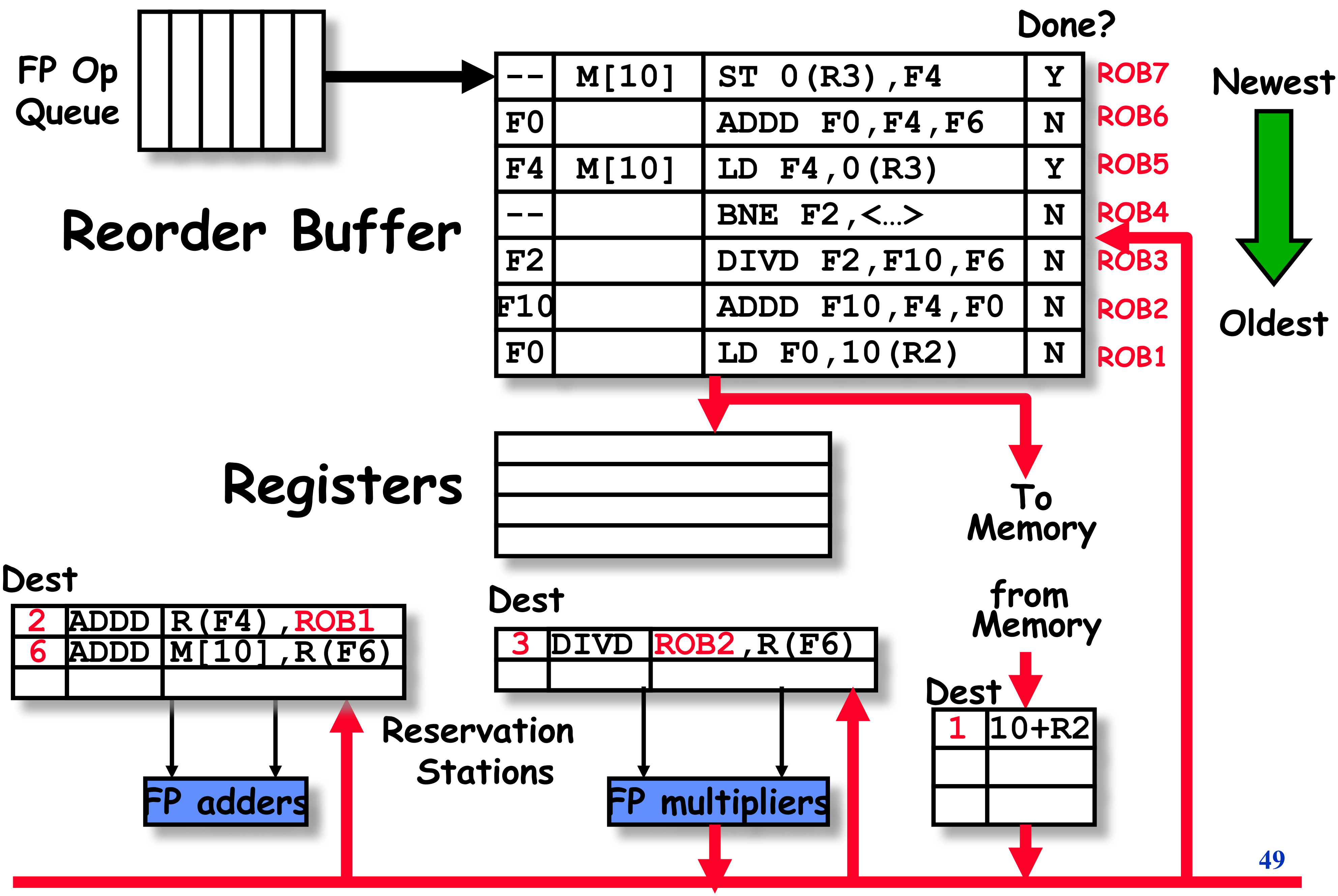
Tomasulo With Reorder buffer:



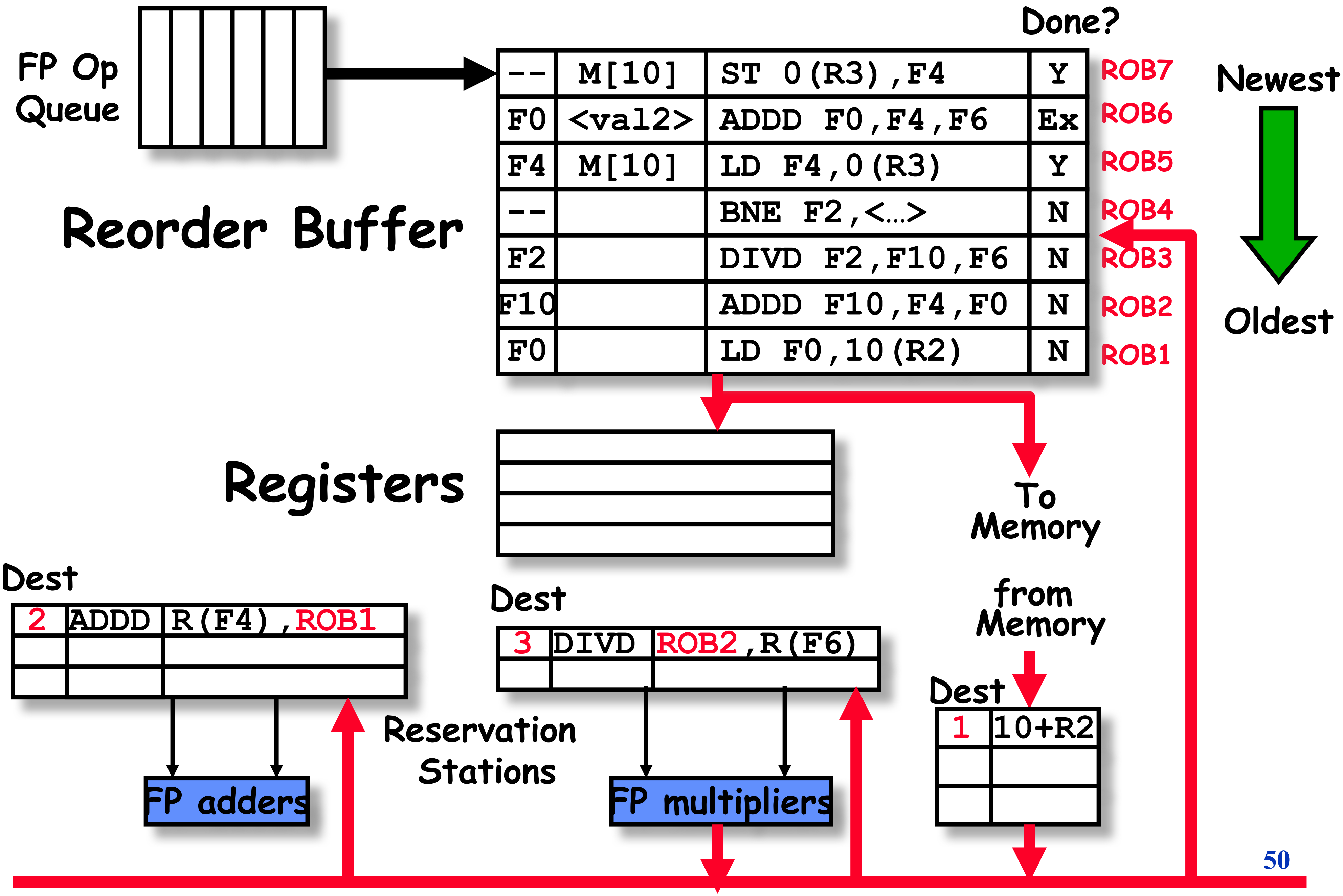
Tomasulo With Reorder buffer:



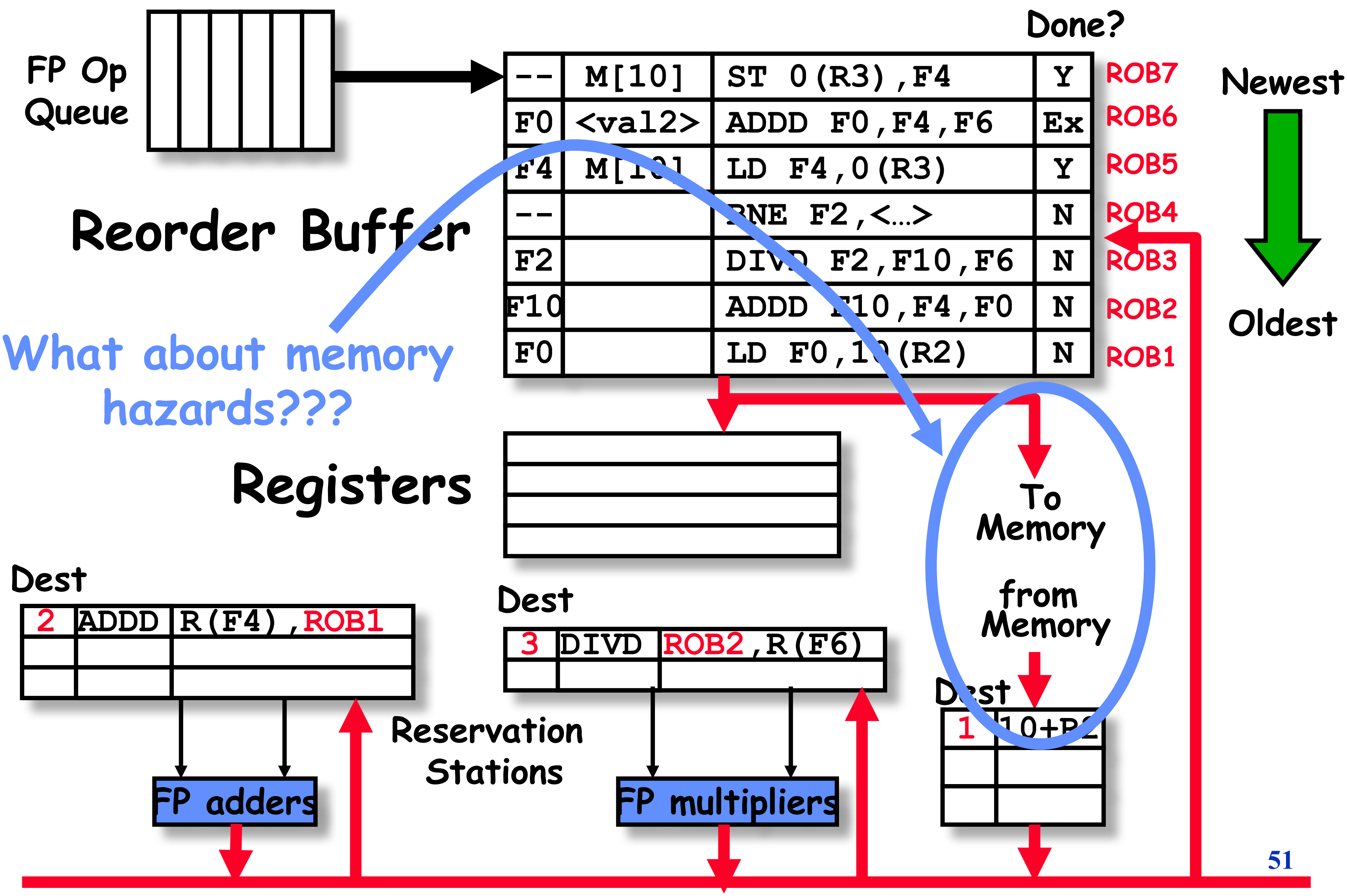
Tomasulo With Reorder buffer:



Tomasulo With Reorder buffer:



Tomasulo With Reorder buffer:



Memory Disambiguation: Sorting out RAW Hazards in memory

- **Question: Given a load that follows a store in program order, are the two related?**
  - (Alternatively: is there a RAW hazard between the store and the load)?

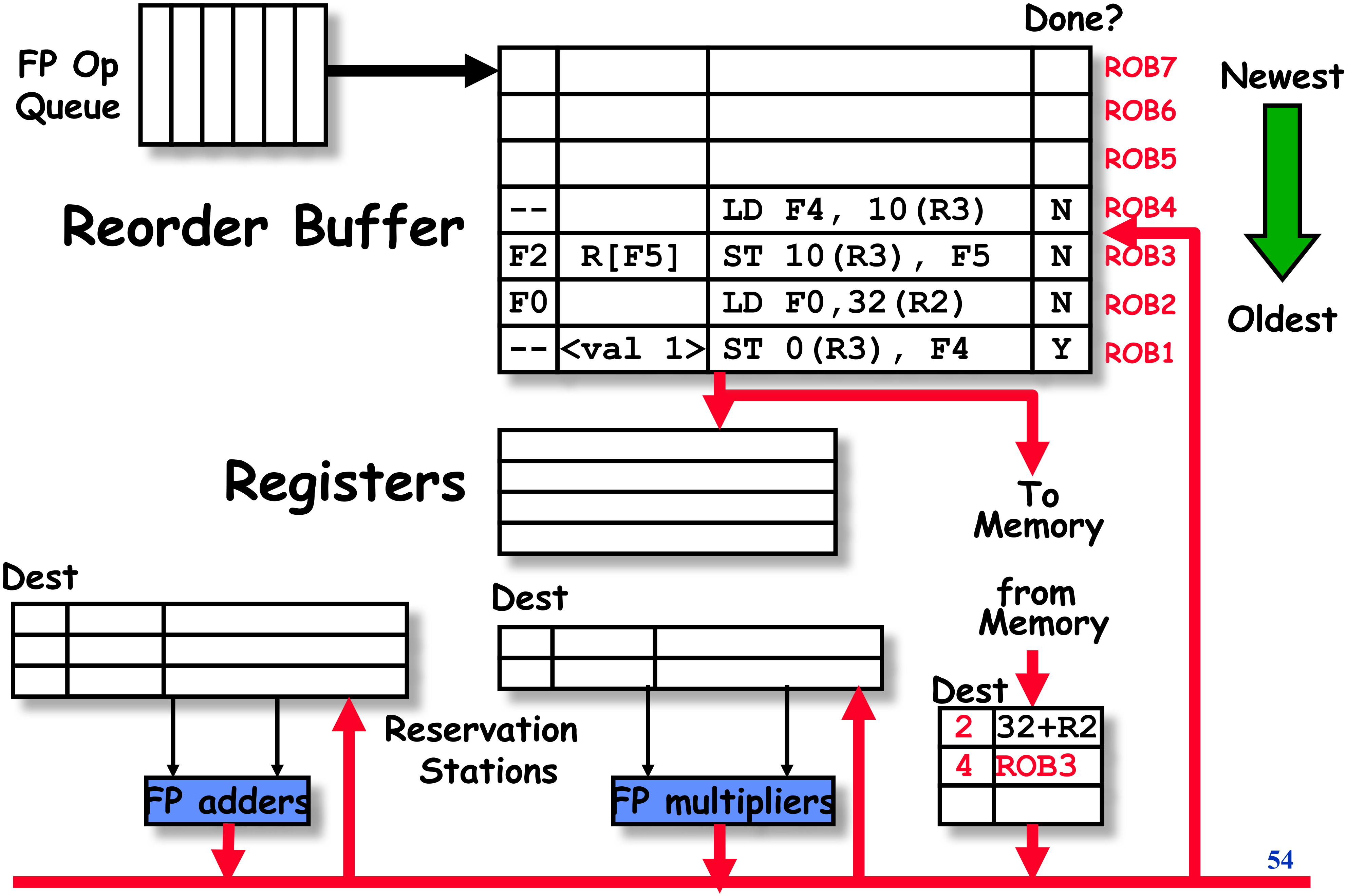
    Eg:
    ```
    st    0(R2),R5
    ld    R6,0(R3)
    ```

- **Can we go ahead and start the load early?**
  - Store address could be delayed for a long time by some calculation that leads to R2 (divide?).
  - We might want to issue/begin execution of both operations in same cycle.
  - Today: Answer is that we are not allowed to start load until we know that address $0(R2) \neq 0(R3)$
  - Next Week: We might guess at whether or not they are dependent (called "dependence speculation") and use reorder buffer to fixup if we are wrong.

Hardware Support for Memory Disambiguation

- **Need buffer to keep track of all outstanding stores to memory, in program order.**
  - Keep track of address (when becomes available) and value (when becomes available)
  - FIFO ordering: will retire stores from this buffer in program order
- **When issuing a load, record current head of store queue (know which stores are ahead of you).**
- **When have address for load, check store queue:**
  - If *any* store prior to load is waiting for its address, stall load.
  - If load address matches earlier store address (associative lookup), then we have a *memory-induced RAW hazard*:
    - » store value available $\Rightarrow$ return value
    - » store value not available $\Rightarrow$ return ROB number of source
  - Otherwise, send out request to memory
- **Actual stores commit in order, so no worry about WAR/WAW hazards through memory.**

Memory Disambiguation:



Relationship between precise interrupts, branch and speculation:

- Speculation is a form of guessing
  – Branch prediction, data prediction
  – 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 precise exceptions!
- Branch prediction is a very important!
  – Need to "take our best shot" at predicting branch direction.
  – If we issue multiple instructions per cycle, lose lots of potential instructions otherwise:
    » Consider 4 instructions per cycle
    » If take single cycle to decide on branch, waste from 4 - 7 instruction slots!
- Technique for both precise interrupts/exceptions and speculation: *in-order completion or commit*
  – This is why reorder buffers in all new processors