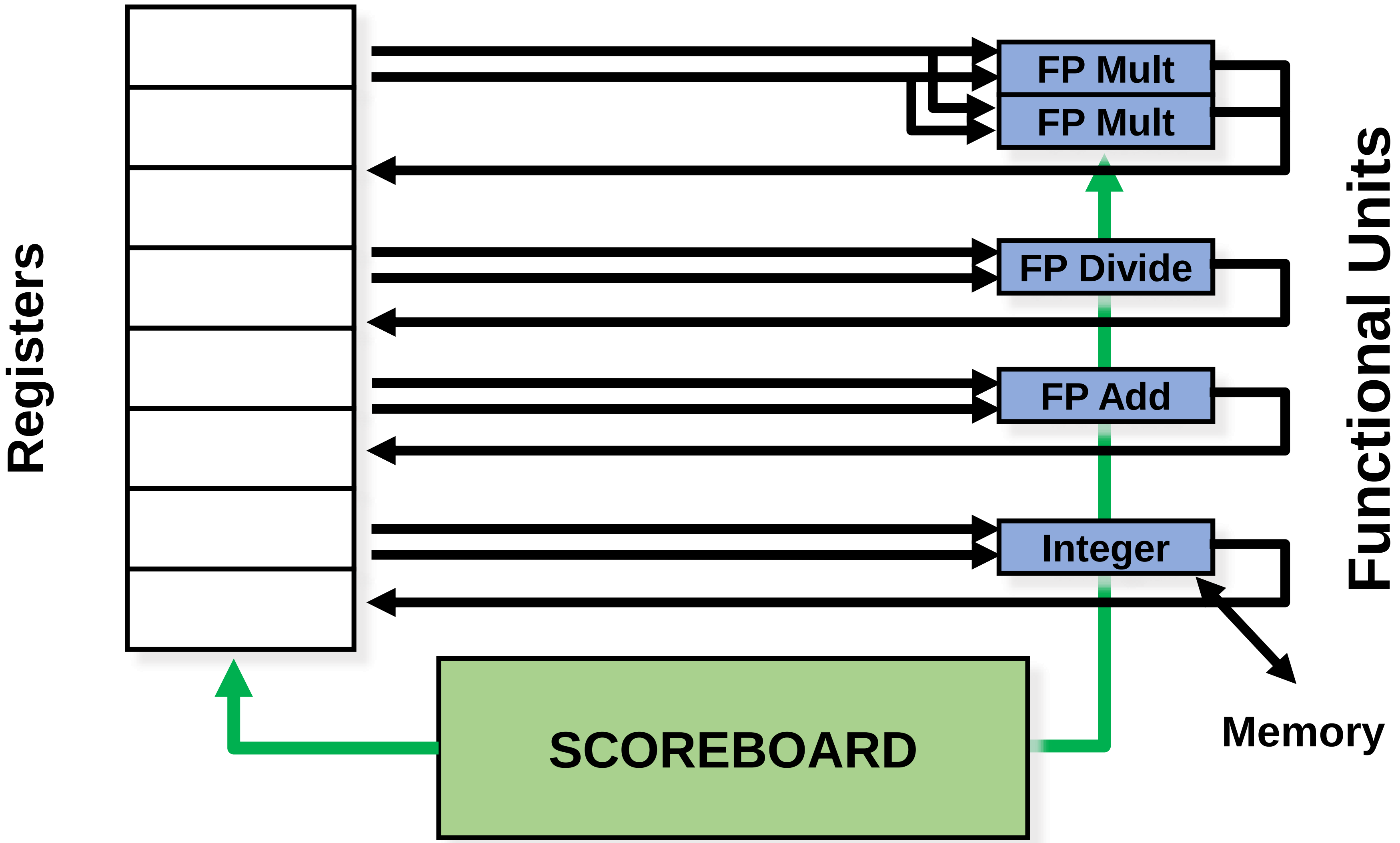
CS425
Computer Systems Architecture

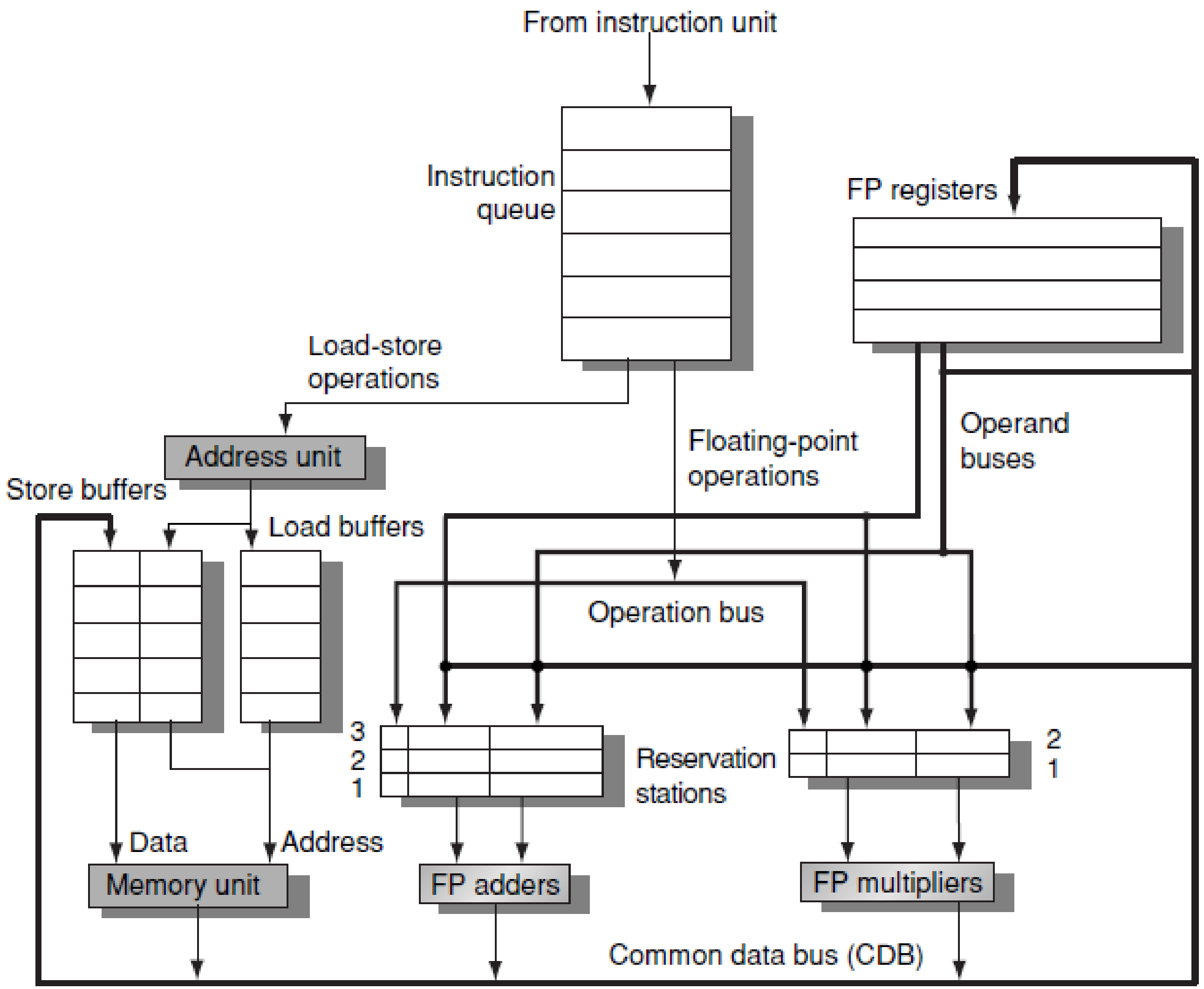
Fall 2019

Re-Order Buffer:
Precise Exceptions and Speculation

Scoreboard Architecture (CDC 6600)



Tomasulo Organization



Tomasulo vs. Scoreboard (IBM 360/91 v. CDC 6600)

Tomasulo	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 them	stall completion
WAW: renaming avoids them	stall issue
Broadcast results from FU	Write/read registers
Control: reservation stations	Central scoreboard

Exception Behavior with ROB

CPI = CPI_{IDEAL} + $Stalls_{STRUC}$ + $Stalls_{RAW}$ + $Stalls_{WAR}$ + $Stalls_{WAW}$ + $Stalls_{CONTROL}$

- Have to maintain:
  - **Data Flow**
  - **Exception Behavior**

Dynamic instruction scheduling (HW)	Static instruction scheduling (SW/compiler)
Scoreboard (reduce RAW stalls)	Loop Unrolling
Register Renaming (reduce WAR & WAW stalls) •**Tomasulo** • **Reorder buffer**	SW pipelining
Branch Prediction (reduce control stalls)	Trace Scheduling
Multiple Issue (CPI < 1) **Multithreading** (CPI < 1)	

Device Interrupt

Network Interrupt

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

Save PC
Disable Interrupts
Supervisor Mode

Restore PC
User Mode

```
Raise priority
Disable All Ints
Save registers
                  ...
lw    r1,20(r0)
lw    r2,0(r1)
addi  r3,r0,#5
sw    0(r1),r3
                  ...
Restore registers
Clear current Int
Re-enable All Ints
Restore priority
RTE
```

Or Could be interrupted by disk

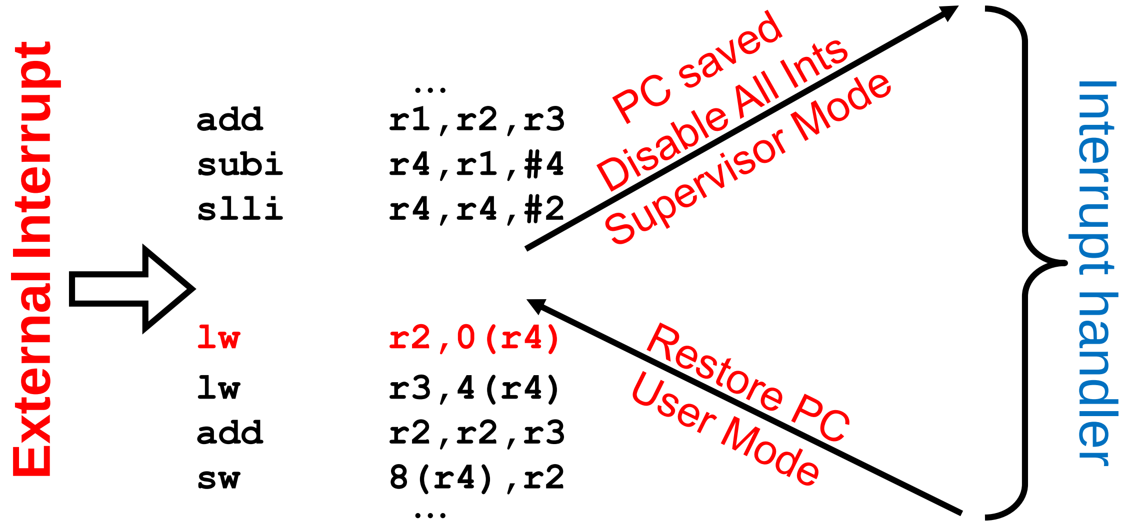
Note that priority must be raised to avoid recursive interrupts!

Types of Interrupts/Exceptions

- I/O device request
- Invoking an operating system service from a user program
- Breakpoint (programmer-requested interrupt)
- Integer arithmetic overflow
- FP arithmetic anomaly
- Page fault (not in main memory)
- Misaligned memory accesses (if alignment is required)
- Memory protection violation
- Using an undefined or unimplemented instruction
- Hardware malfunctions
- Power failure

Precise Interrupts/Exceptions

- An interrupt or exception is precise if there is an instruction (or interrupt point) for which:
  - All instructions before this instruction have fully completed
  - None of the instructions after this instructions (including the interrupting instruction) has modified the machine state
- This means that we can restart the execution from the interrupt point and still “get the correct results”
  - In the example: the Interrupt point is the `lw` instruction



Imprecise Interrupt/Exception

- An exception is imprecise if the processor state when an exception is raised does not look **exactly** as if the instructions were executed sequentially in strict program order

- Occurrence in two possibilities:
  - The pipeline may have already completed instructions that are later in program order
  - The pipeline may have not yet completed some instructions that are earlier in program order

Precise interrupt point requires multiple PCs when there are delayed branches

```
       addi  r4,r3,#4
       sub   r1,r2,r3
PC:    bne   r1,there
PC+4:  and   r2,r3,r5
       <other insts>
```

Interrupt point described as <PC, PC+4>

```
       addi  r4,r3,#4
       sub   r1,r2,r3
PC:    bne   r1,there
PC+4:  and   r2,r3,r5
       <other insts>
```

Interrupt point described as:

<PC+4, there> (branch was taken)
or
<PC+4, PC+8> (branch was not taken)

Why do we need precise interrupts?

- Several interrupts/exceptions need to be restartable
  - i.e. TLB faults. Fix translation and then restart the faulting load/store
  - IEEE gradual underflow, illegal operation,

e.g: $$f(x) = \frac{\sin(x)}{x}$$

$$x \to 0 \qquad f(0) = \frac{0}{0} \Rightarrow NaN + illegal\_operation$$

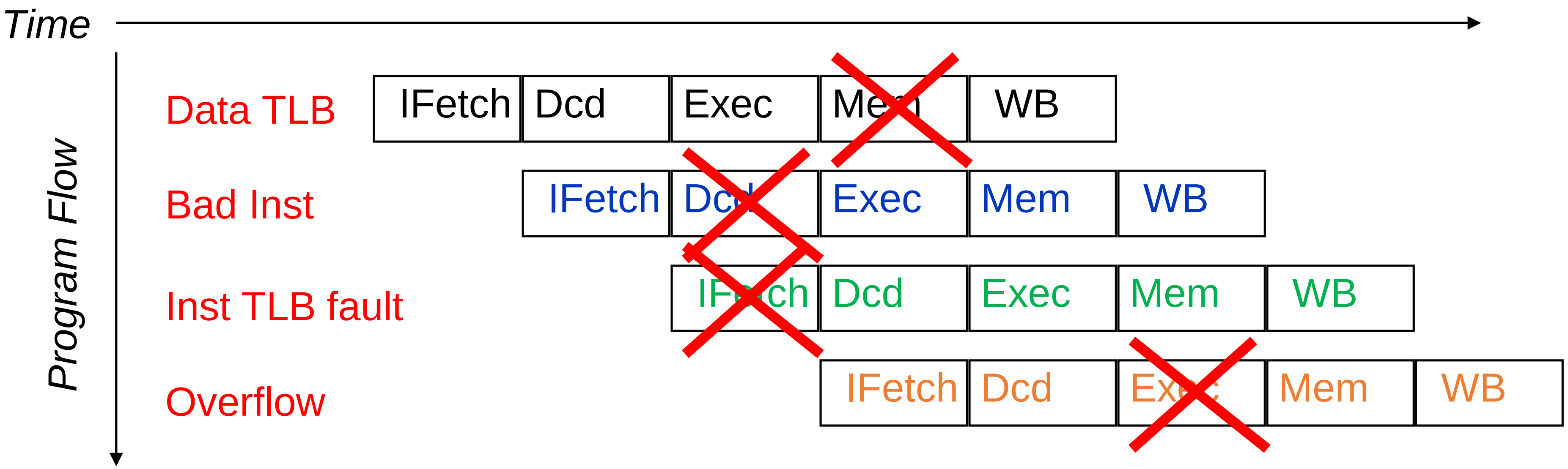
Want to take exception, replace *NaN* with 1, then restart.

- Restartability does not require *preciseness*. However, preciseness makes restarts *much simpler*
- Simplifies the Operating System (OS)
  - Less state needs to be saved away if unloading process.
  - Quick to restart (for fast interrupts)

Precise Exceptions in 5-stage DLX

- Exceptions may occur in different stages of the processor pipeline (i.e. out of order):
  - Arithmetic exceptions occur in execution stage
  - TLB faults can occur in instruction fetch or memory stage
- How do we guarantee precise exceptions? Mark the instructions with an “exception status” and wait until the WB stage to take the exception
  - Interrupts are marked as NOPs (like bubbles) that are placed into pipeline instead of an instruction.
  - Assume that interrupt condition persists in case NOP flushed
  - Clever instruction fetch might start fetching instructions from interrupt vector, but this is complicated and needs to switch to supervisor mode, saving of one or more PCs, etc

Another look at the exception problem



- Use the pipeline!
  - Each instruction has an exception status field.
  - Keep the PCs for every instruction in the pipeline.
  - Check the exception status when the instruction reaches the WB stage
- When an instruction reaches the WB stage and has an exception:
  - Save PC $\Rightarrow$ EPC, Interrupt vector addr $\Rightarrow$ PC
  - Convert all fetched instructions to NOPs
- It works because of in-order completion/WB

Scoreboard Example: Cycle 62

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

In-order issue

Out-of-order execute

Out-of-order commit!

Functional unit status:

Time	Name	Busy	Op	dest Fi	S1 Fj	S2 Fk	FU Qj	FU Qk	Fj? Rj	Fk? Rk
	Integer	No								
	Mult1	No								
	Mult2	No								
	Add	No								
	Divide	No								

Register result status:

Clock		F0	F2	F4	F6	F8	F10	F12	...	F30
62	*FU*									

Tomasulo Example: Cycle 57

Instruction status:

Instruction		j	k	Issue	Exec Comp	Write Result
LD	F6	34+	R2	1	3	4
LD	F2	45+	R3	2	4	5
MULTD	F0	F2	F4	3	15	16
SUBD	F8	F6	F2	4	7	8
DIVD	F10	F0	F6	5	56	57
ADDD	F6	F8	F2	6	10	11

	Busy	Address
Load1	No	
Load2	No	
Load3	No	

Reservation Stations:

Time	Name	Busy	Op	S1 Vj	S2 Vk	RS Qj	RS Qk
	Add1	No					
	Add2	No					
	Add3	No					
	Mult1	No					
	Mult2	Yes	DIVD	M*F4	M(A1)		

In-order issue

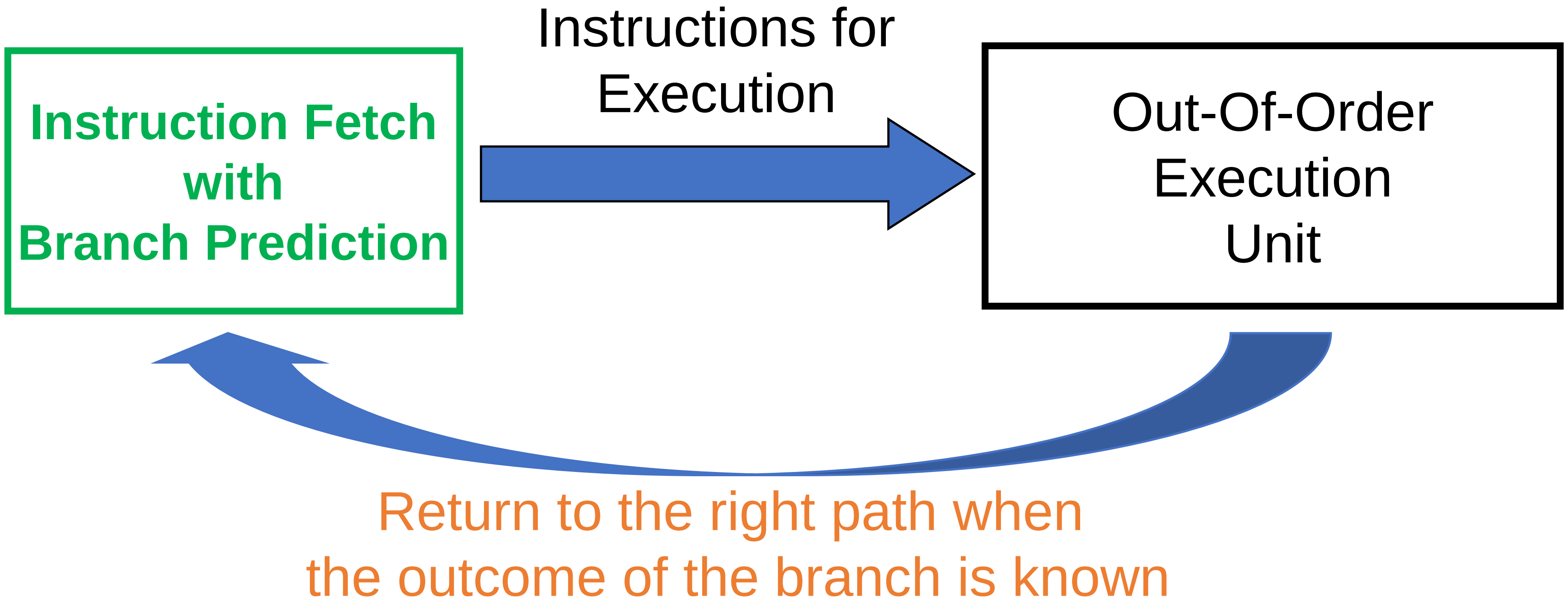
Out-of-order execute

Out-of-order commit!

Register result status:

Clock		F0	F2	F4	F6	F8	F10	F12	...	F30
56	*FU*						Result			
	Reg File	M*F4	M(A2)		(M-M+M)	(M-M)				

Issue: “Fetch” unit



- Instructions from a potentially mispredicted branch path have been already executed.
- Instruction fetch decoupled from execution

Branch must execute fast for loop overlap!

- In the loop-unrolling example, we assume that the branches are executed from a "fast" integer unit to achieve overlap!

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

- What happens if the branch depends on the outcome of MULTD?
  - We lose all benefits
  - We have to predict the outcome of the branch
  - If we predict "taken" the prediction would be correct most of the time.

Prediction: Branches, Dependencies, Data

- Branch Prediction is necessary for good performance
- We studied branches in the previous lecture. Modern architectures now predict many things: data dependencies, actual data, and results of groups of instructions

- Why does prediction work?
  - Underlying algorithm has regularities.
  - Data that is being operated on has regularities.
  - Instruction sequence has redundancies that are artifacts of way that humans/compilers think about problems.

Problem: Out-of-Order Completion

- Scoreboard and Tomasulo operate as follows:
  - In-order issue, out-of-order execution, out-of-order completion

- We need a way to synchronize the completion stage of instructions with the program order (i.e. with issue-order)
  - Easiest way is with in-order completion (i.e. re-order buffer)
  - Other Techniques (Smith paper): Future File, History Buffer

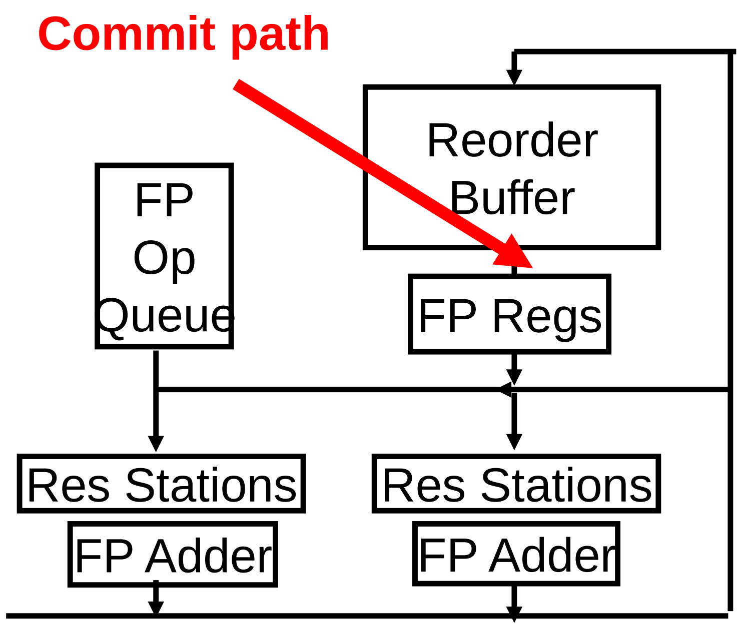
Precise Interrupts and Speculation:

- During the Issue stage of instructions we operate as if as we are predicting that all previous instructions do not generate exceptions
  - Branch prediction, data prediction
  - If we speculate and are wrong, need to back up and restart execution to the point at which we predicted incorrectly
  - This is exactly same as precise exceptions!

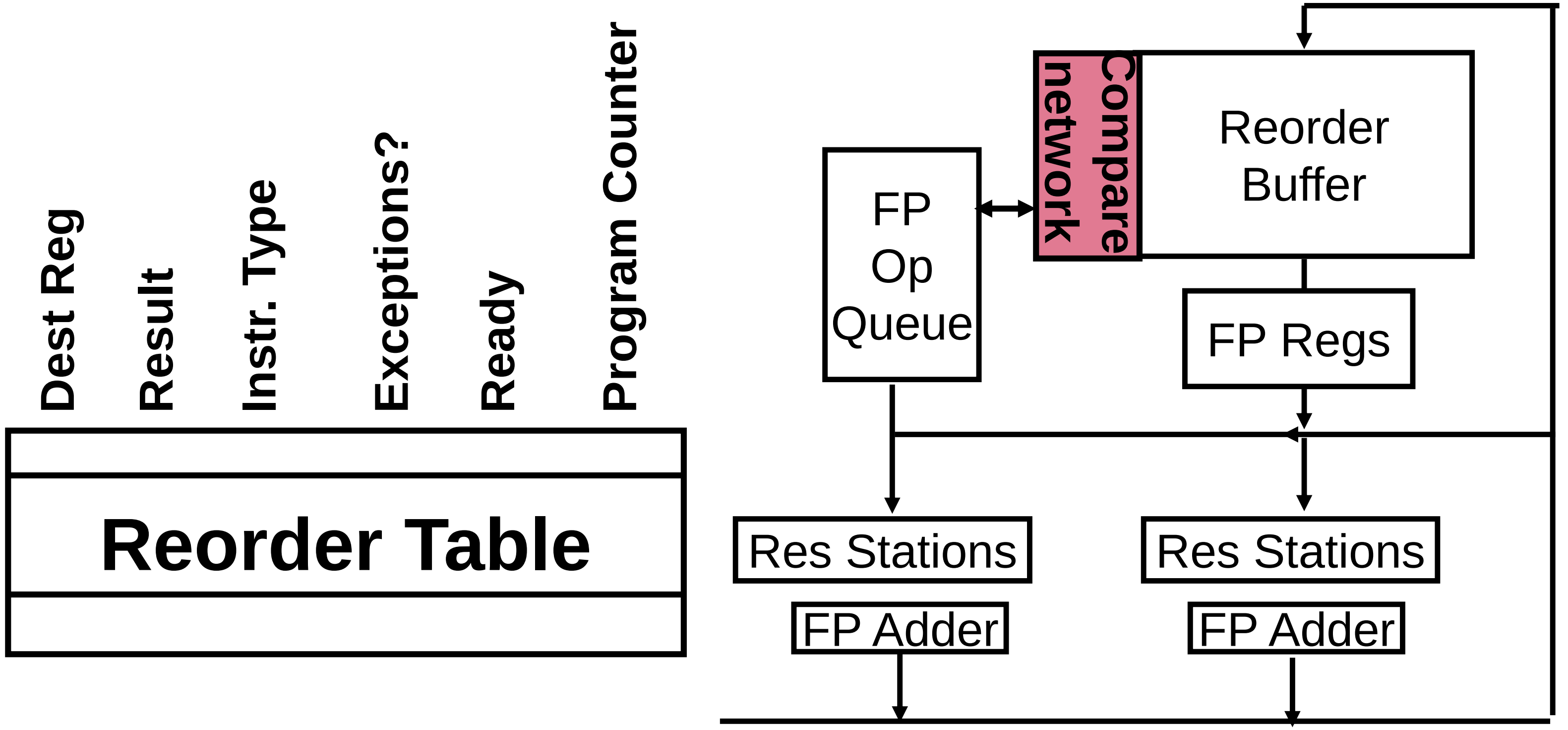
- Common technique for precise interrupts/exceptions and speculation: **in-order completion or commit**
  - All modern out-of-order processors typically use a form of re-order buffer (ROB)

HW support for precise interupts/exceptions

- Idea behind Reorder Buffer (ROB): keep the instructions in a FIFO, with the exact order that they are issued.
  - Each ROB entry contains PC, dest reg/mem, result, exception status
- When an instruction completes execution then the results are placed in the allocated entry in the ROB.
  - Supplies operands to other instruction between execution complete & commit ⇒ more registers like RS
  - Tag results with ROB buffer number instead of reservation station
- The instructions change the machine state at the commit stage not on the WB ⇒ in order commit ⇒ values at head of ROB are placed in registers
- This technique allows us to cancel/squash speculatively executed instructions during mispredicted branches or exceptions

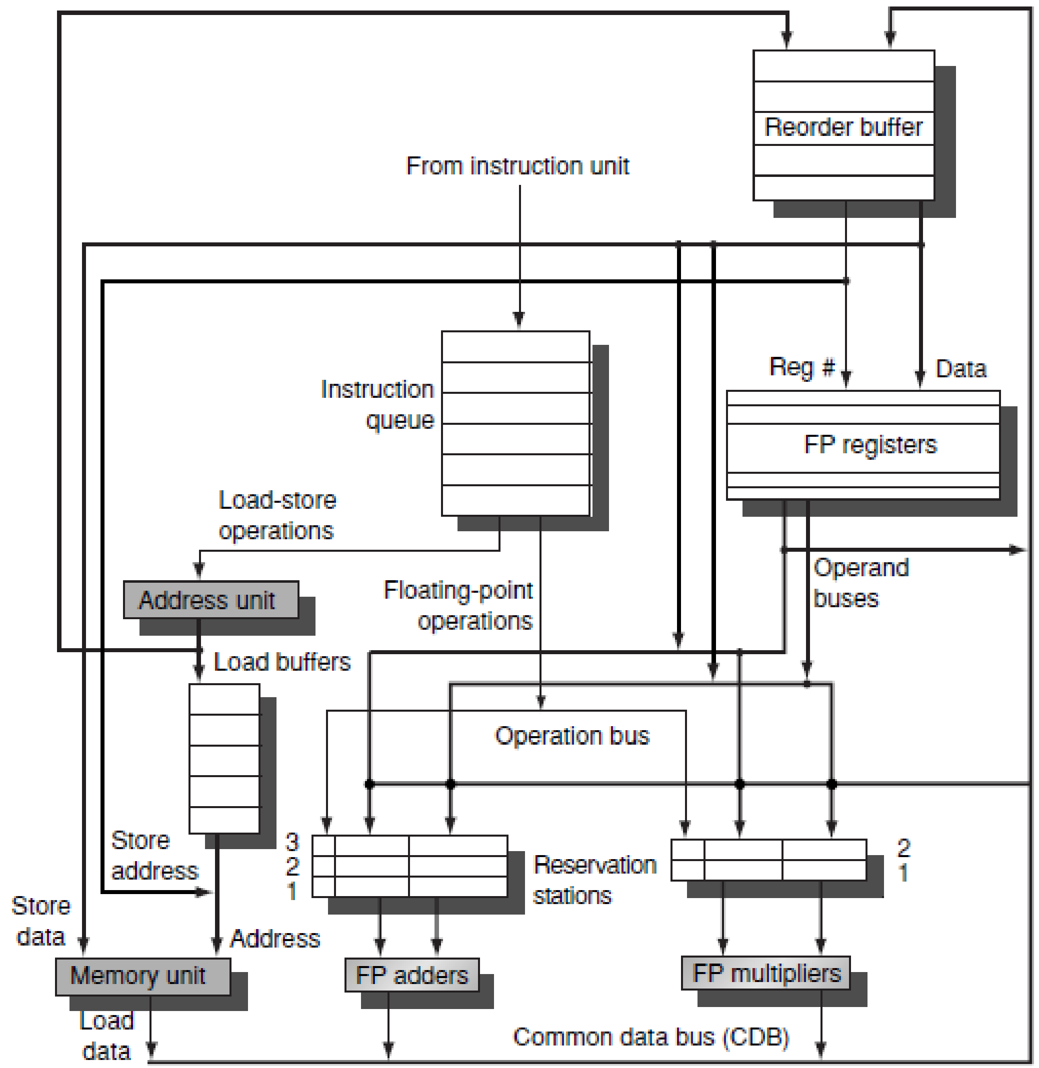


HW Support for Reorder Buffer (ROB)?



- How do we find the last "version" of each register ?
- Multi-ported ROB like the register file
- Integrate store buffer into ROB since we have in order commit. Stores use Result field for ROB tag until data ready on CDB.
- Can we also integrate the reservation stations ?

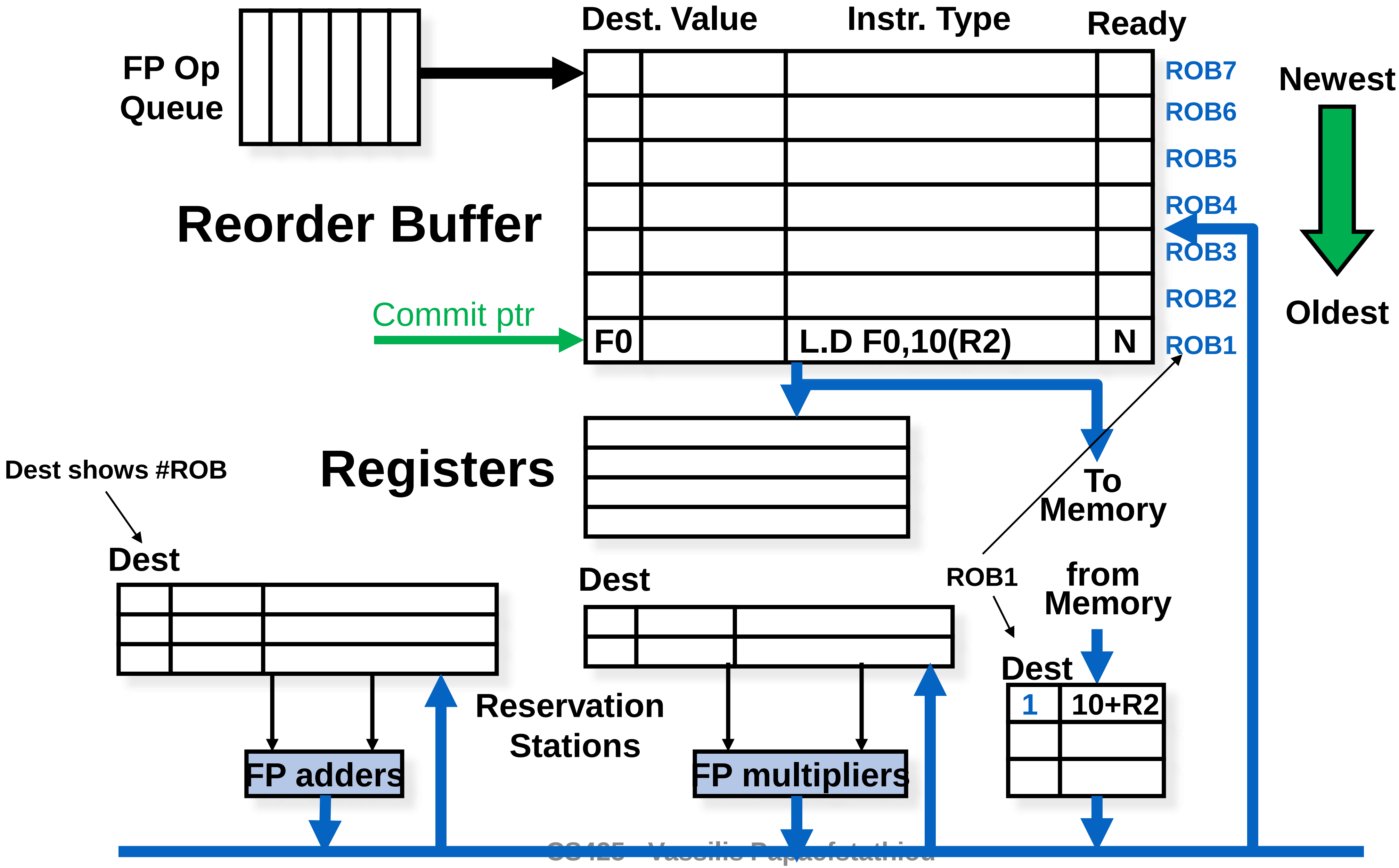
Tomasulo with ROB: Basic Block Diagram



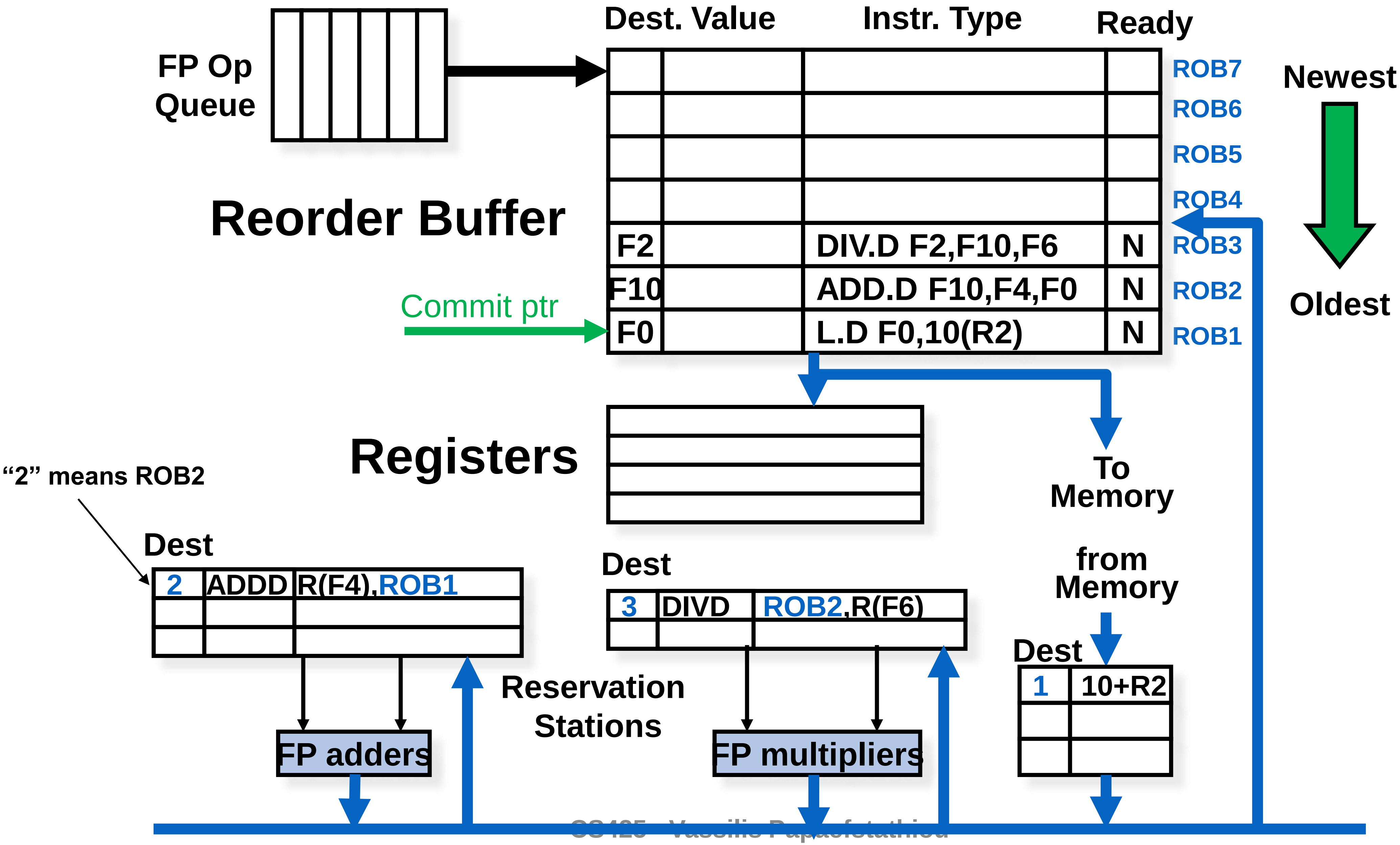
Four Stages of Tomasulo with ROB

1. Issue: Get Instruction from Op Queue
   – If there are free reservation stations and reorder buffer slot, issue instr & send operands & reorder buffer no. for destination (sometimes called “dispatch”)
2. Execution: Execute the Instruction in the Execution Unit (EX)
   – When the values of the 2 source regs are ready then execute the instruction; otherwise, watch CDB for result; when both in reservation station, execute; checks RAW (“issue”)
3. Write result: End of Execution (WB)
   – Write on Common Data Bus to all awaiting FUs & reorder buffer; mark reservation station available.
4. Commit: Update the dst reg with the value from the reorder buffer
   – 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 or exception flushes reorder buffer. (also called “graduation” or “retirement”)

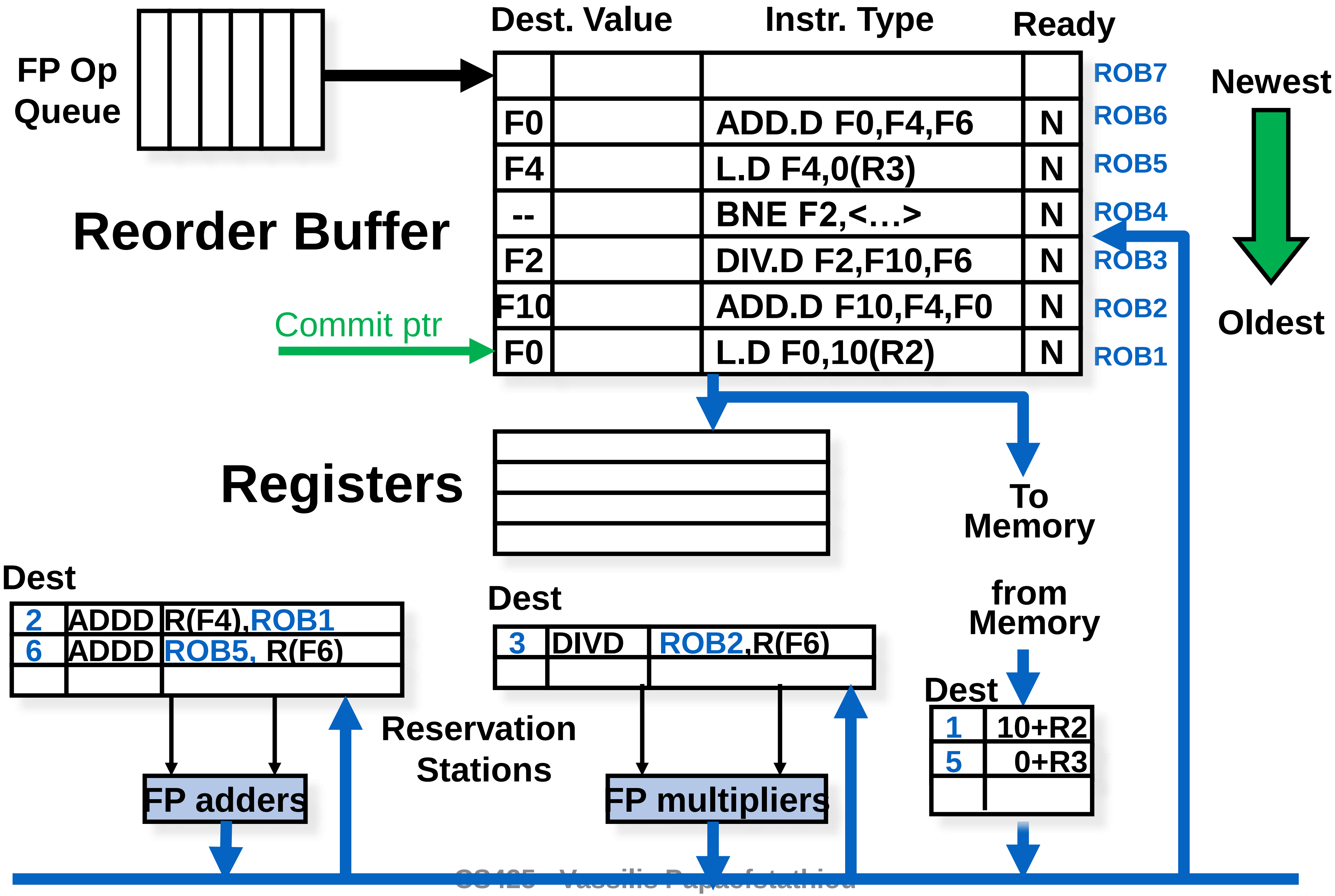
Tomasulo With Reorder buffer



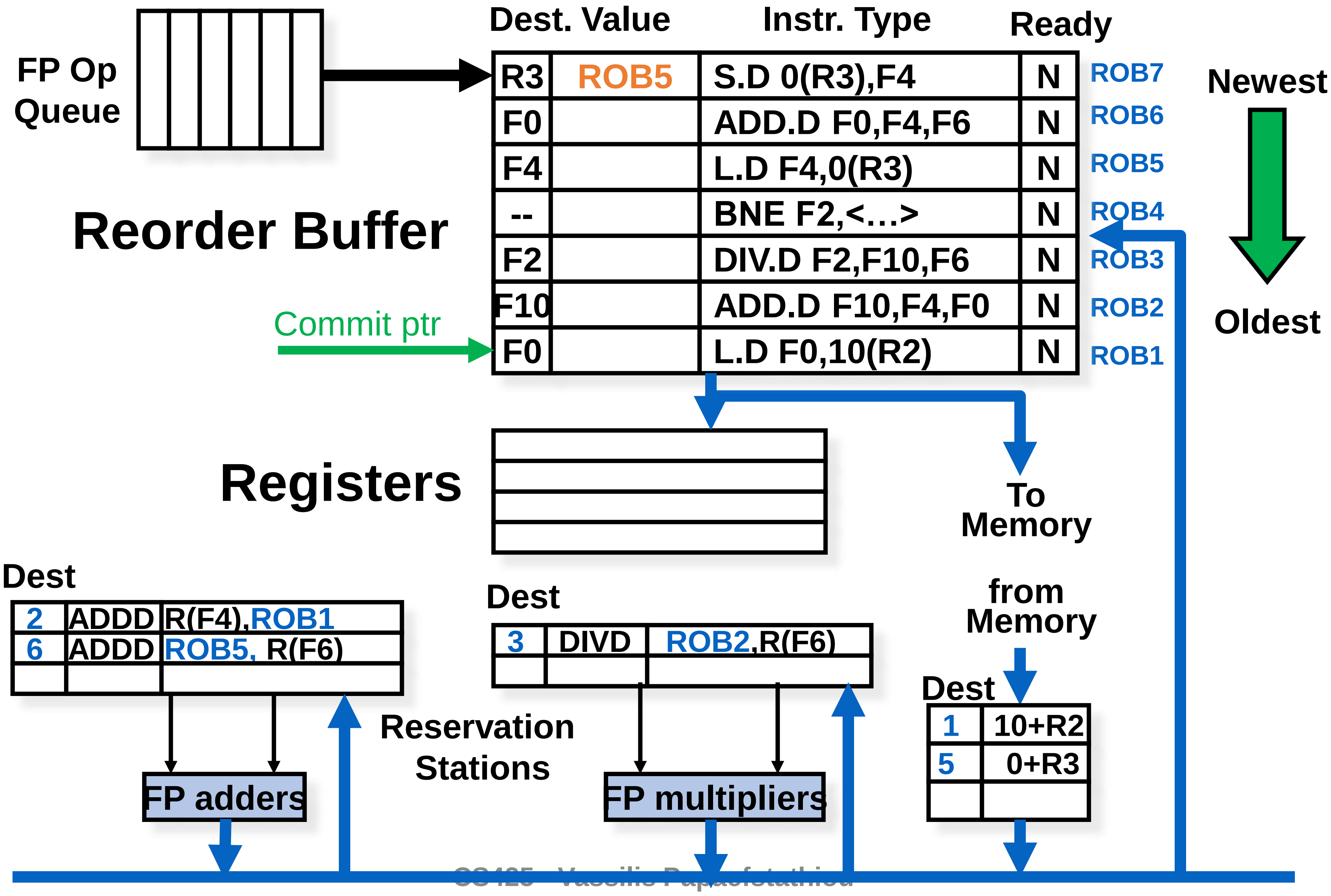
Reorder buffer (after 2 cycles)



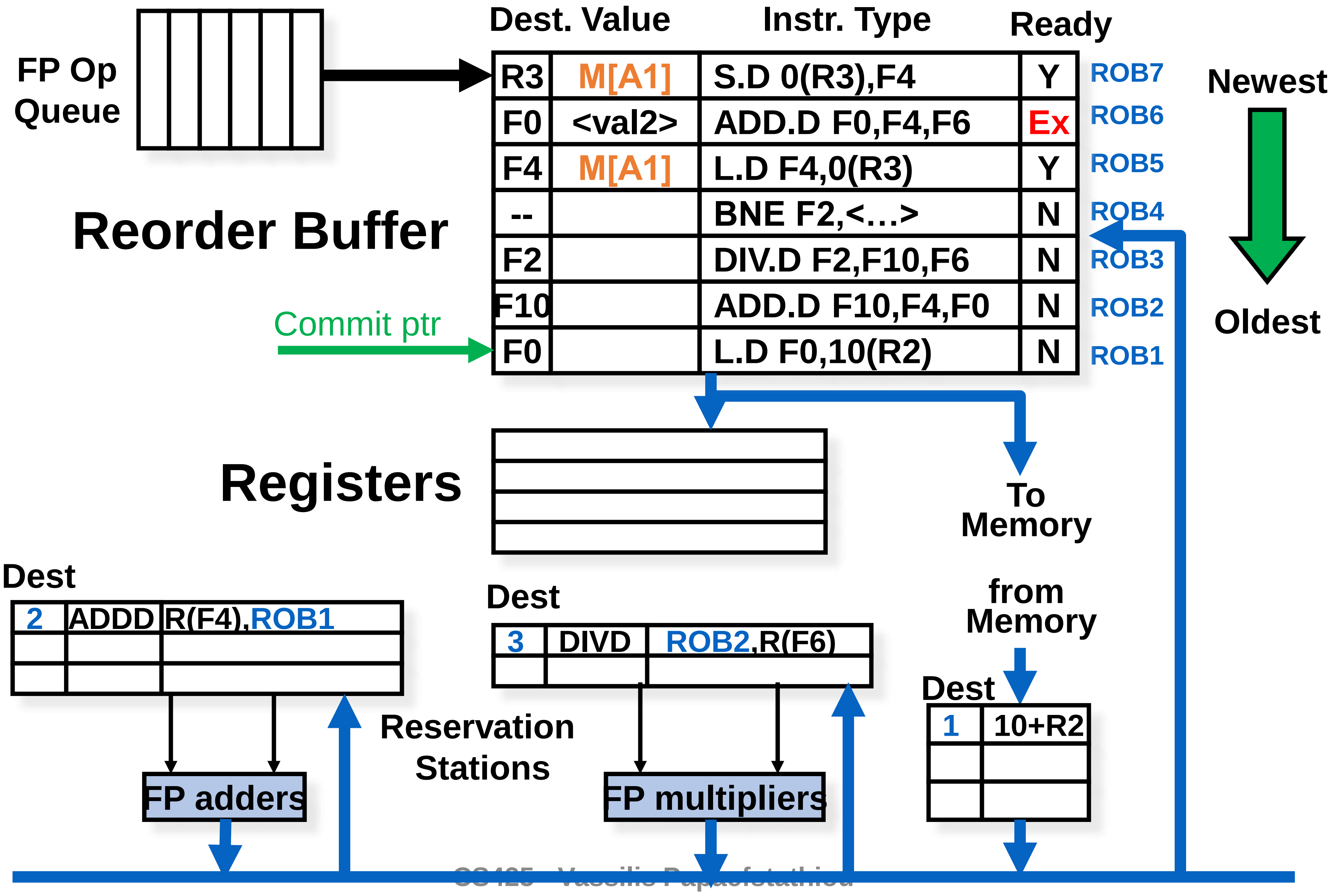
Reorder buffer (after 3 cycles)



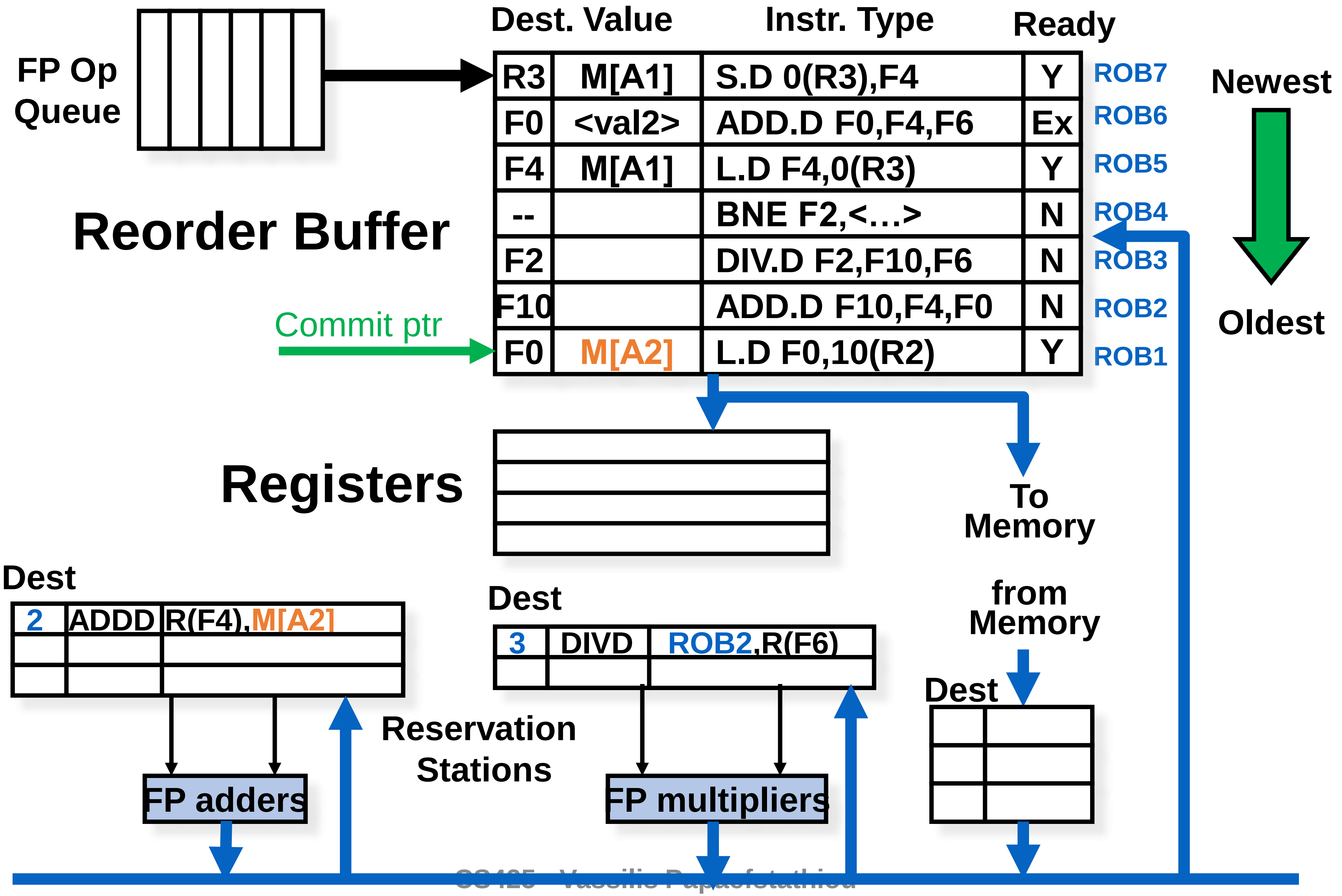
Reorder buffer (after 1 cycle)



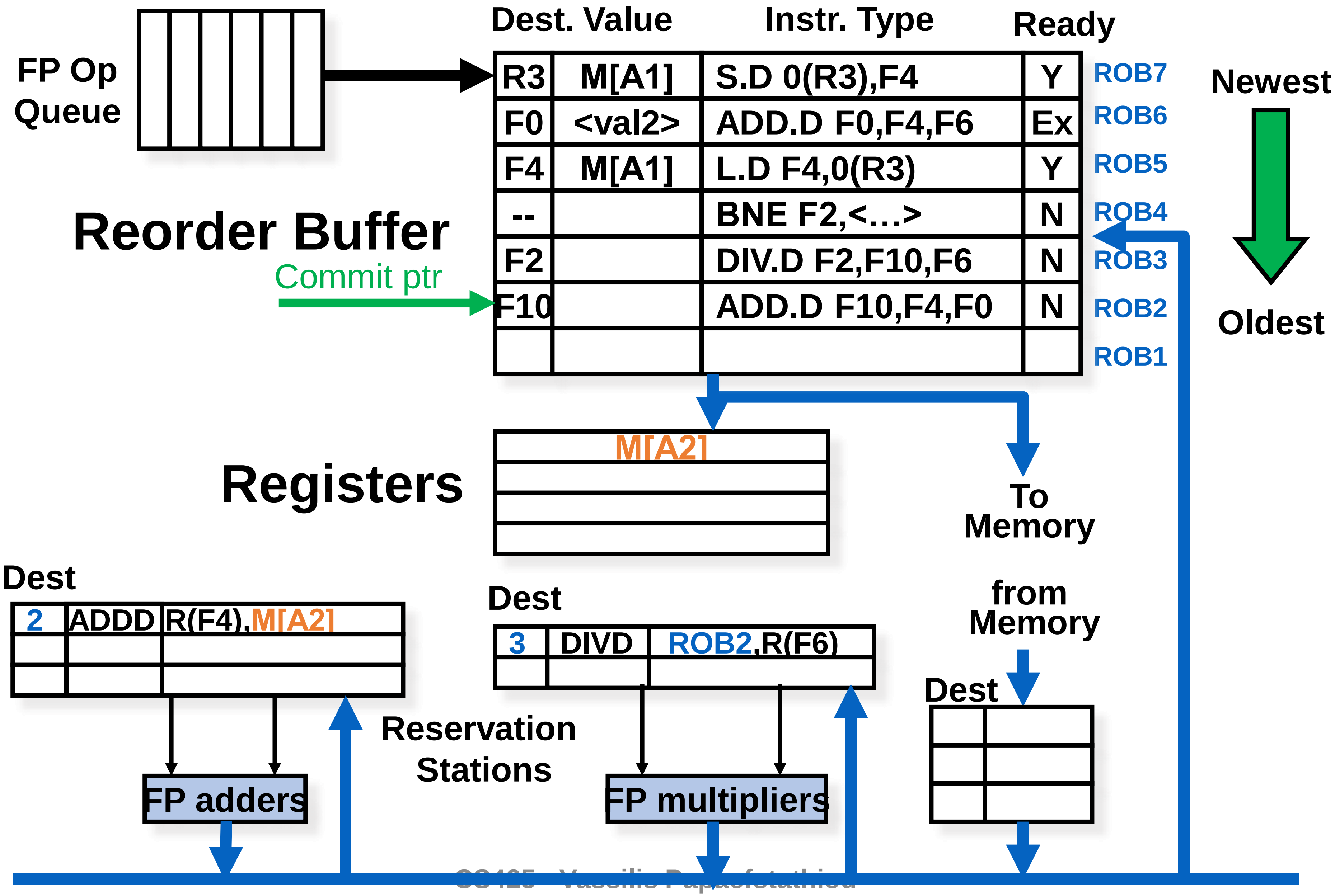
Tomasulo With Reorder buffer



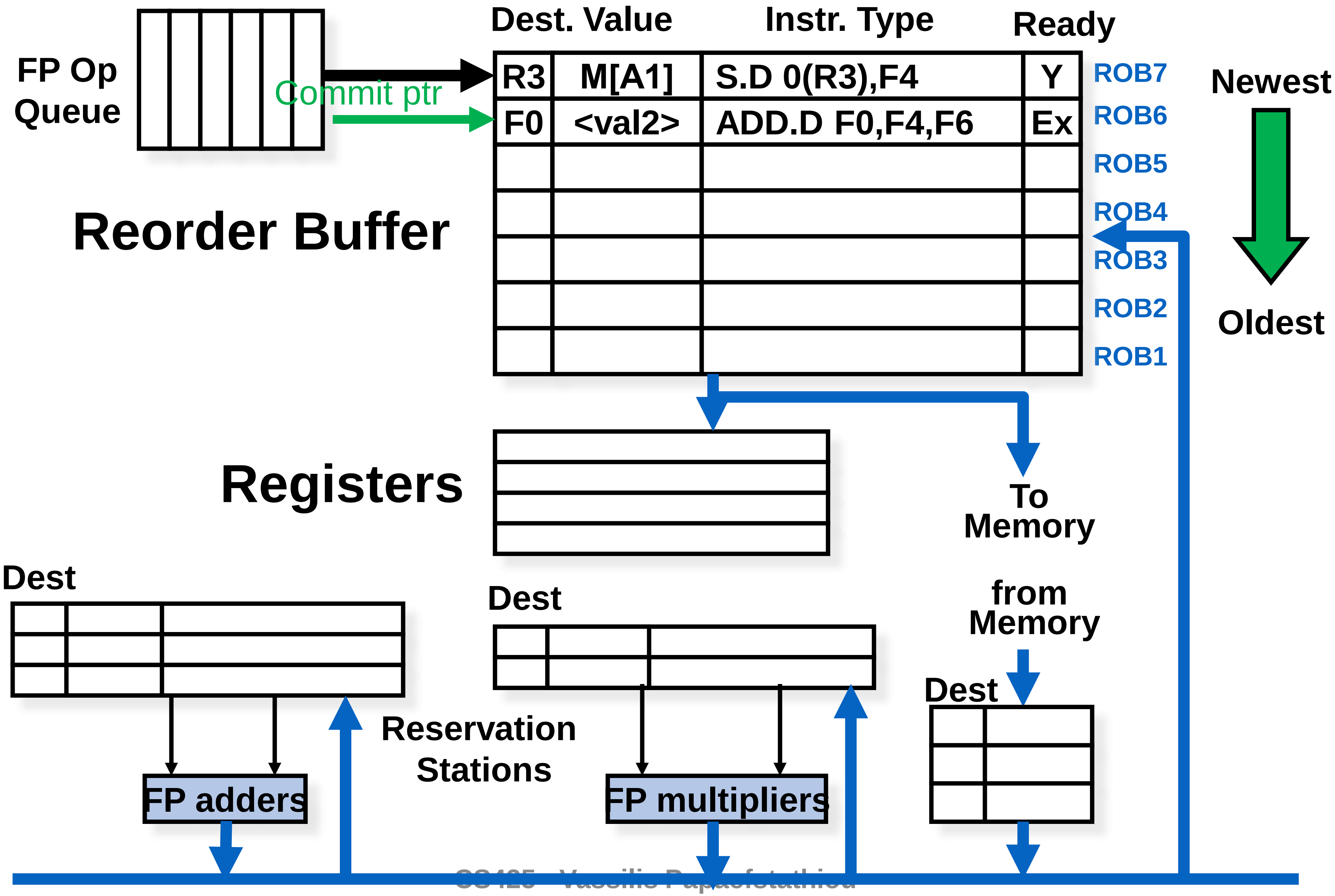
Tomasulo With Reorder buffer



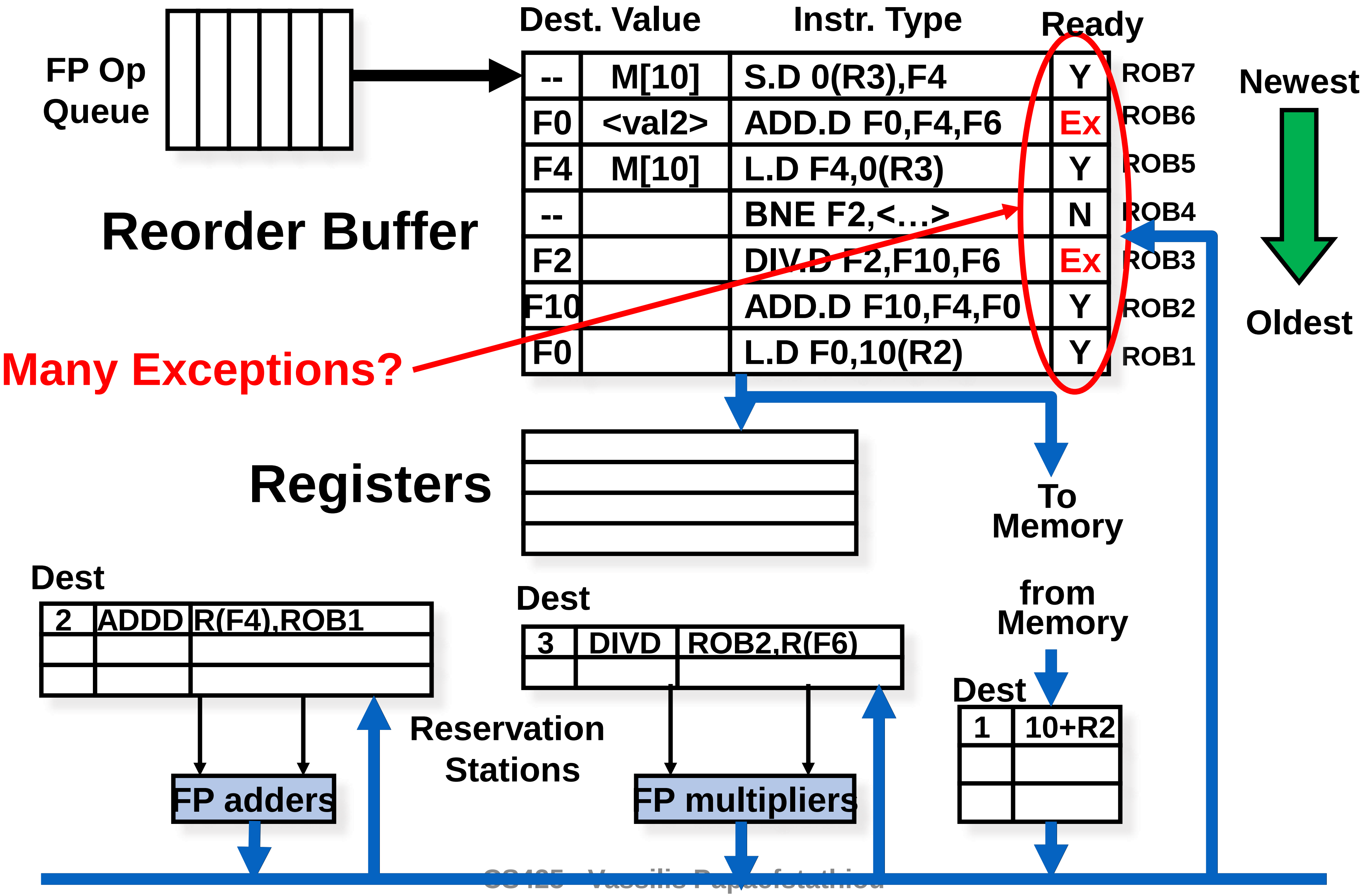
Tomasulo With Reorder buffer



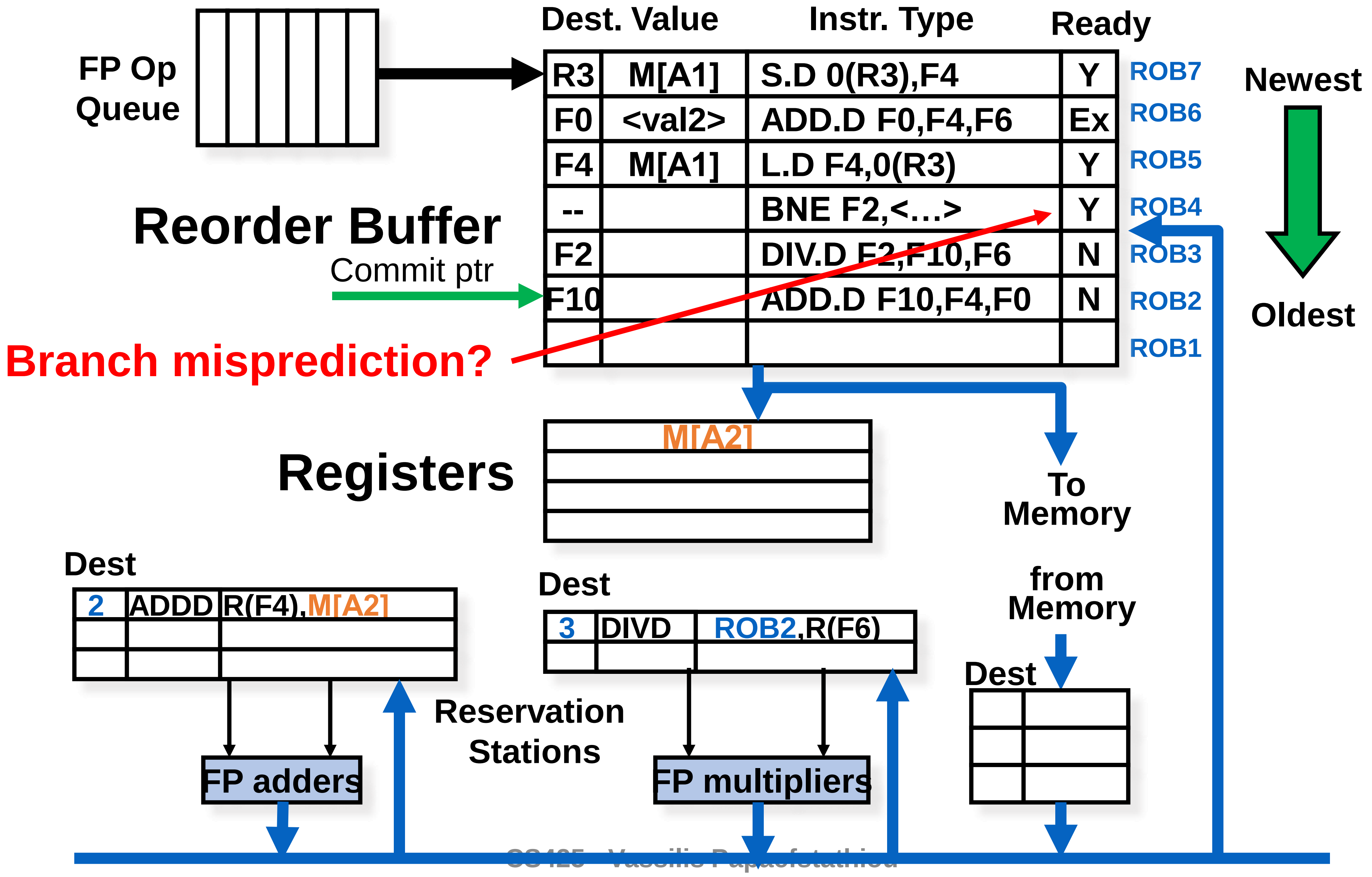
Tomasulo With Reorder buffer



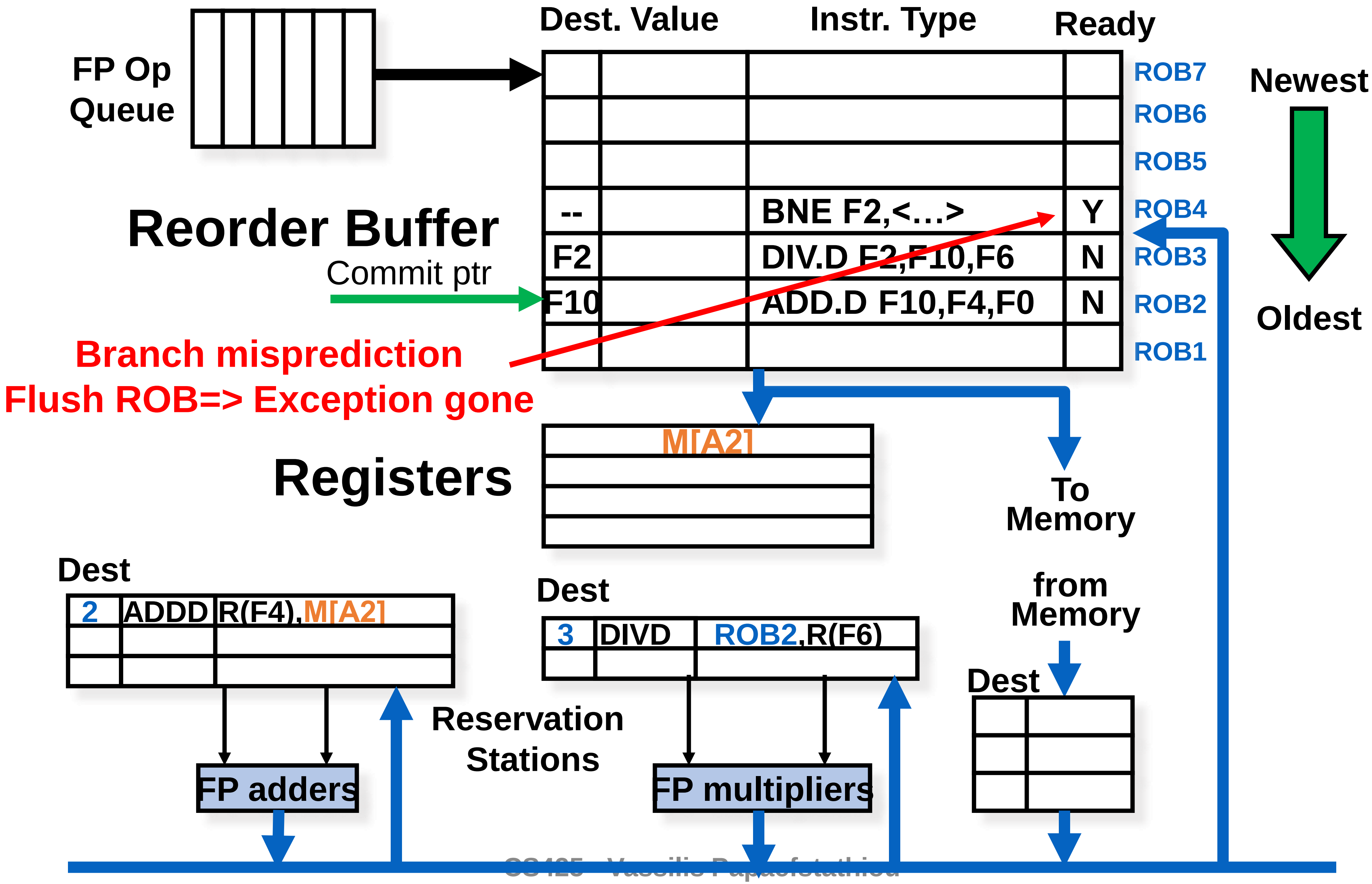
Reorder buffer: Precise Exceptions



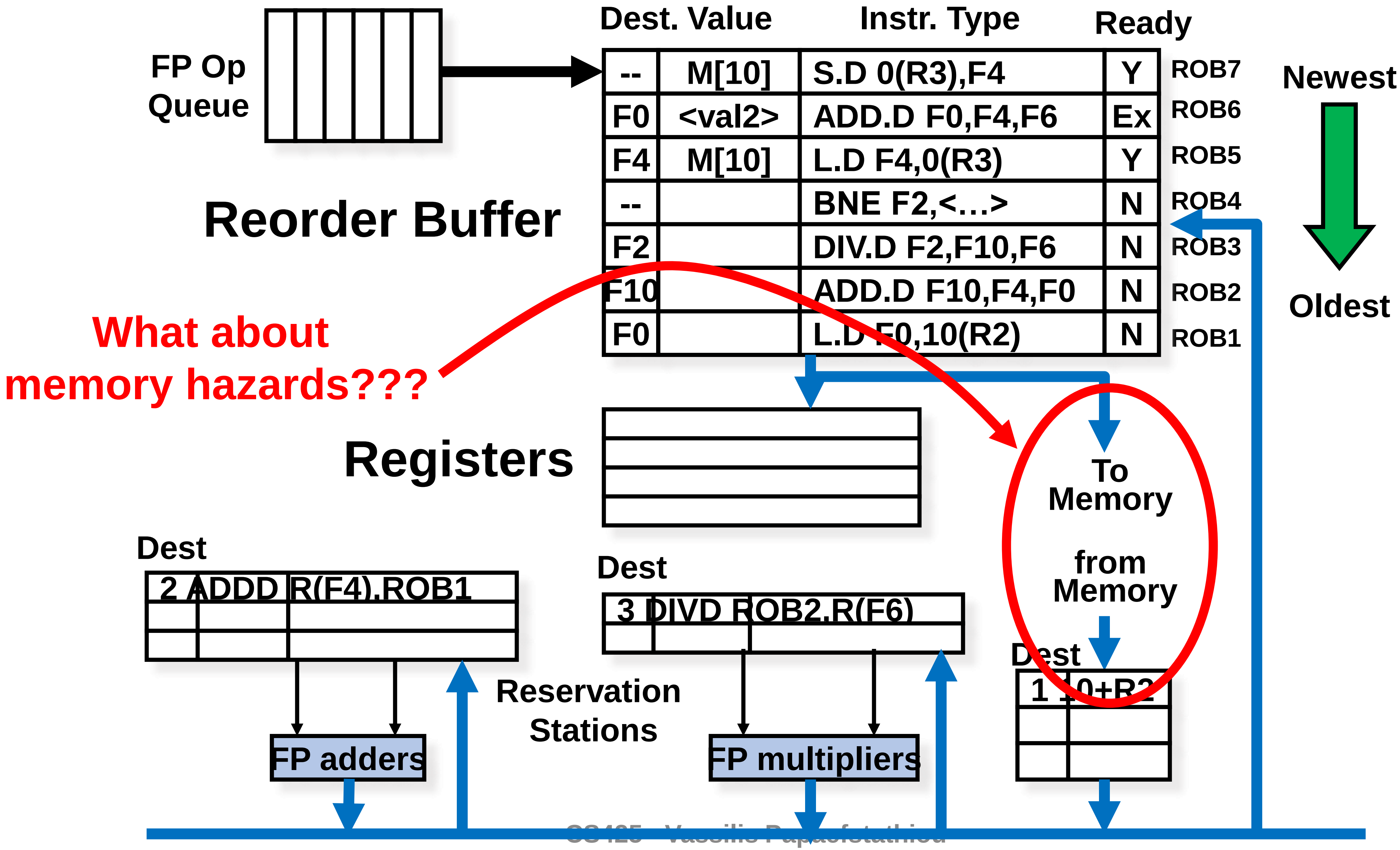
Reorder buffer: Branch Misprediction



Reorder buffer: Branch Misprediction



Tomasulo With Reorder buffer



Memory Disambiguation: WAW/WAR Hazards

- Like Hazards in Register File, we must avoid hazards through memory:
  - WAW and WAR hazards through memory are eliminated with speculation because the actual updating of memory occurs in order, when a store is at the head of the ROB, and hence, no earlier loads or stores can still be pending.

Memory Disambiguation: RAW Hazards

- Challenge: Given a load that follows a store in program order, are these two related?
  - What if there is a RAW hazard between the store and the load?
    Eg:

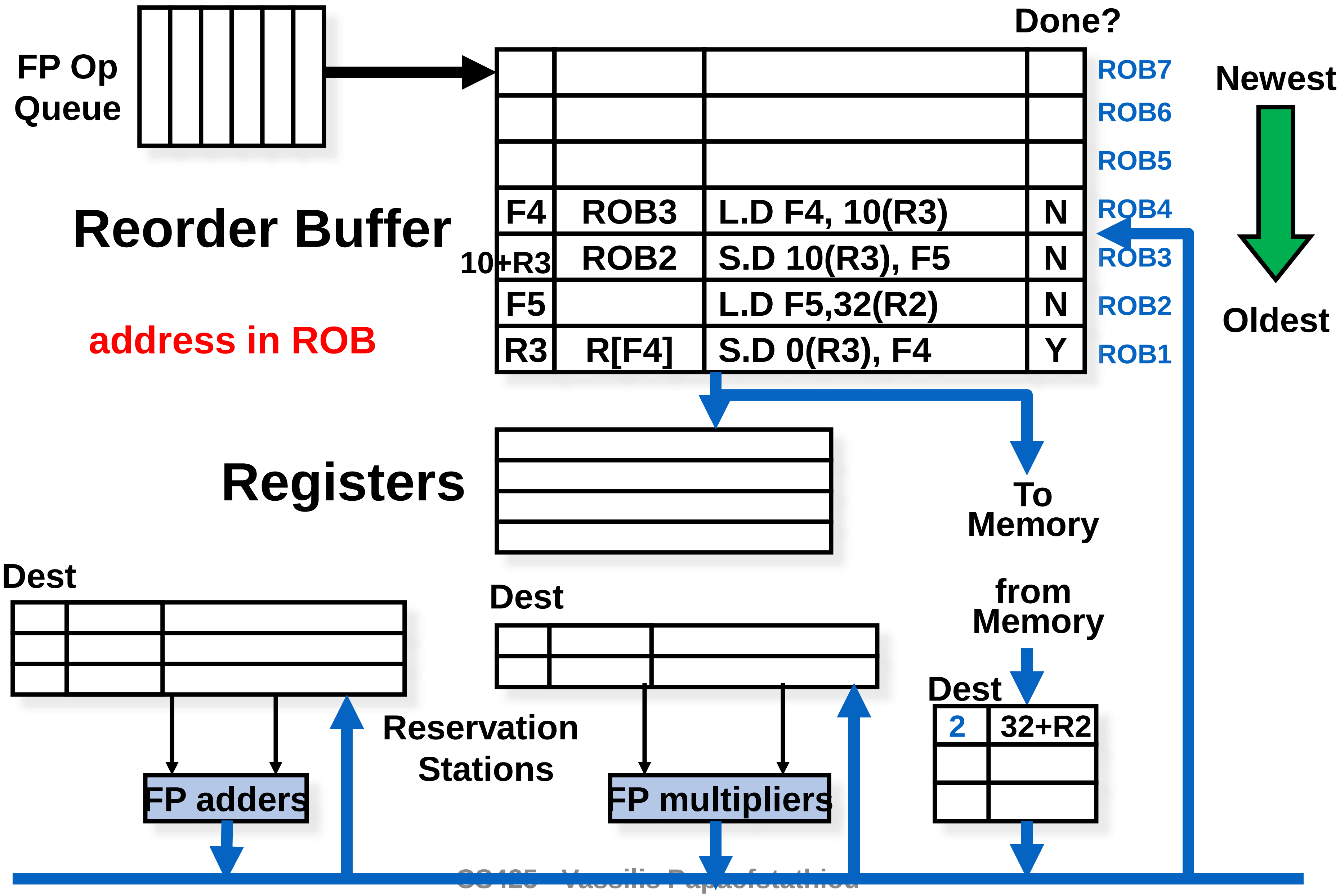
```
st    0(R2),R5
ld    R6,0(R3)
```

- Can we proceed and issues the load to the memory system?
  - Store address could be delayed for a long time by some calculation that leads to R2 (e.g. divide).
  - We might want to issue/begin execution of both operations in same cycle.
  - **Solution1:** Answer is that we are not allowed to start load until we know that address 0(R2) $\neq$ 0(R3)
  - **Solution2:** We might guess at whether or not they are dependent (called "dependence speculation") and use reorder buffer to fixup if we are wrong.

HW support for Memory Disambiguation

- Store buffer keeps all pending 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 the head of the store buffer (which stores precede)
- When we have the address of the load, check the buffer:
  - 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
- Stores commit in order, there are no WAW/WAR hazards in memory.

Memory Disambiguation



Register Renaming

```
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
```

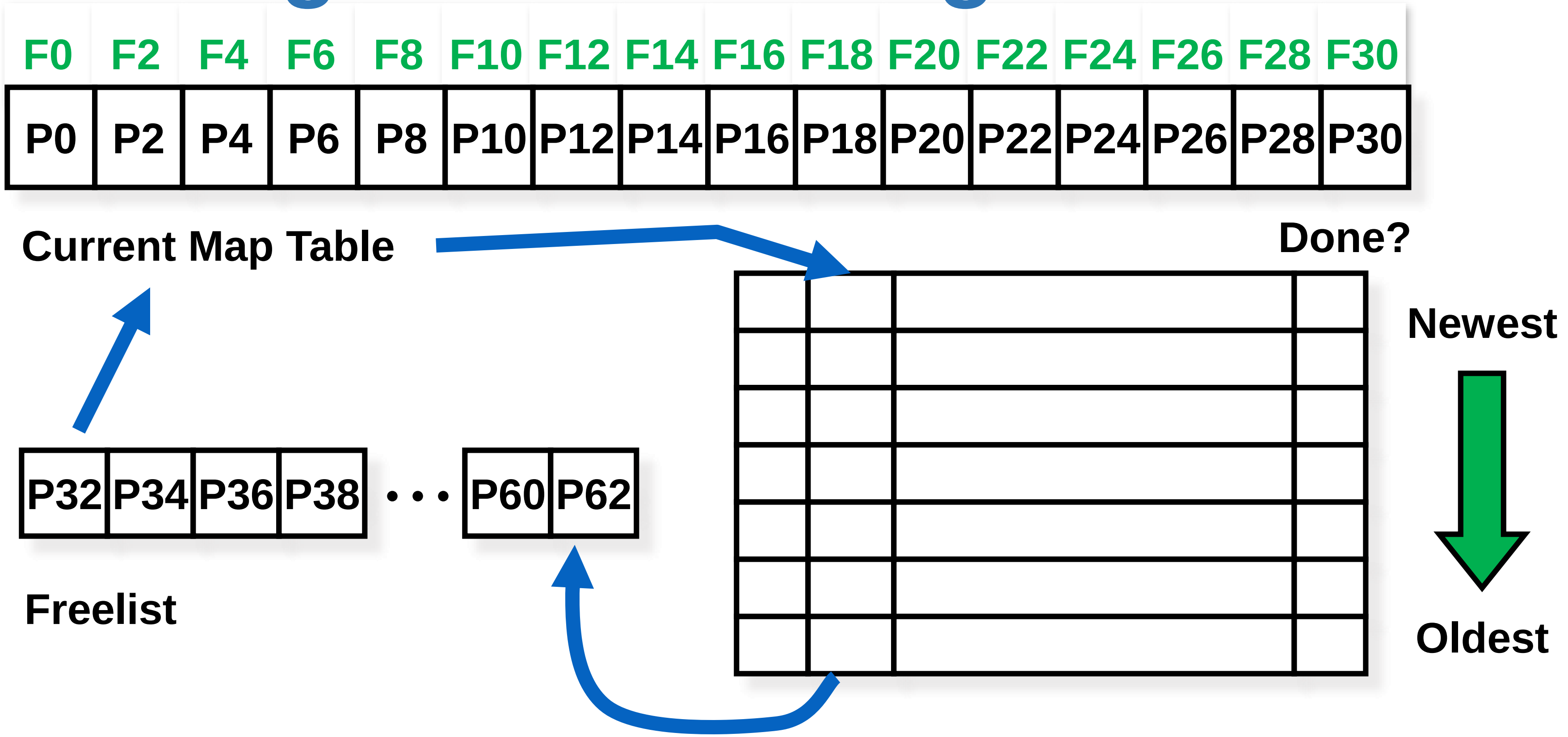
WAW WAR

register renaming

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

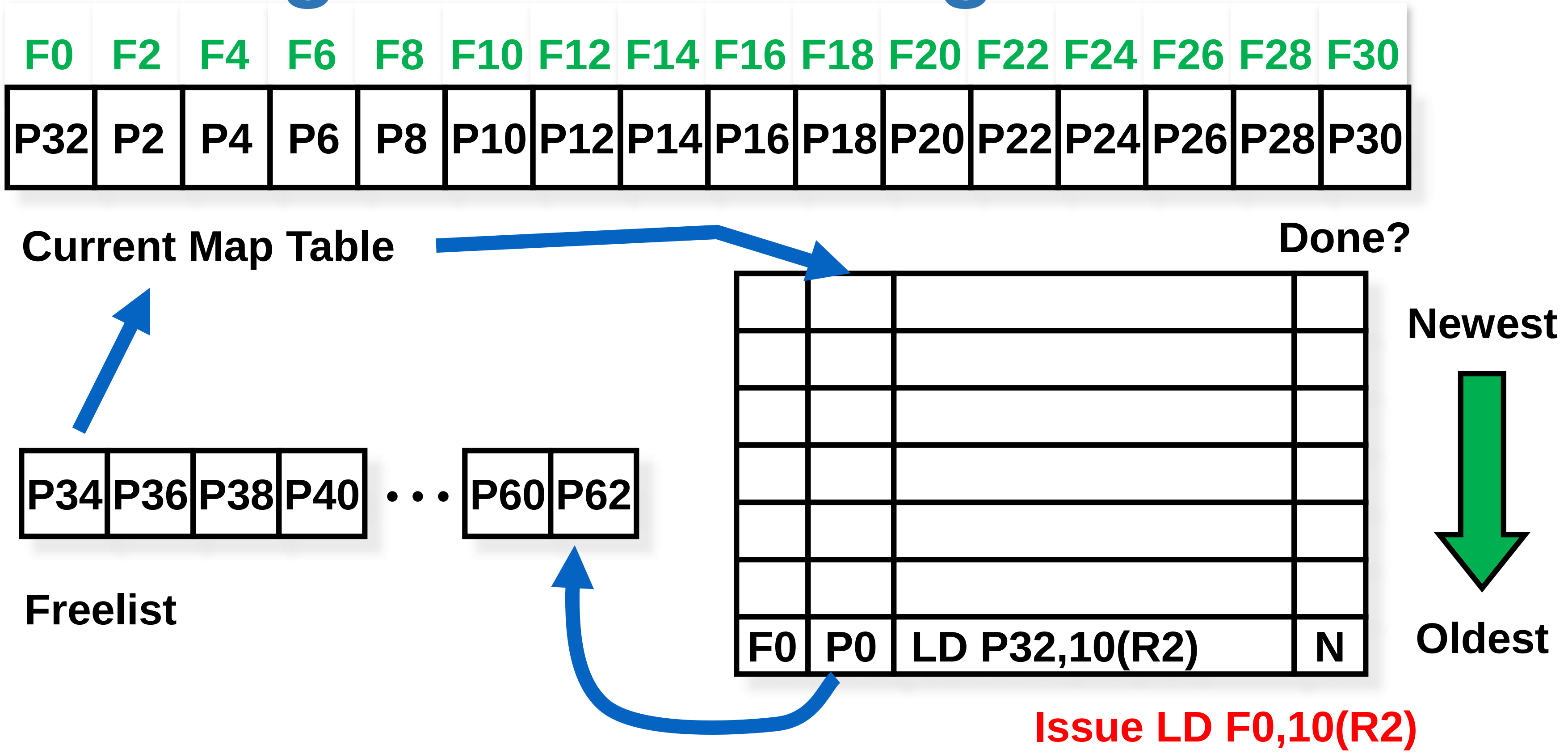
- What happens with branches?
- Tomasulo can handle renaming across branches

Explicit register renaming



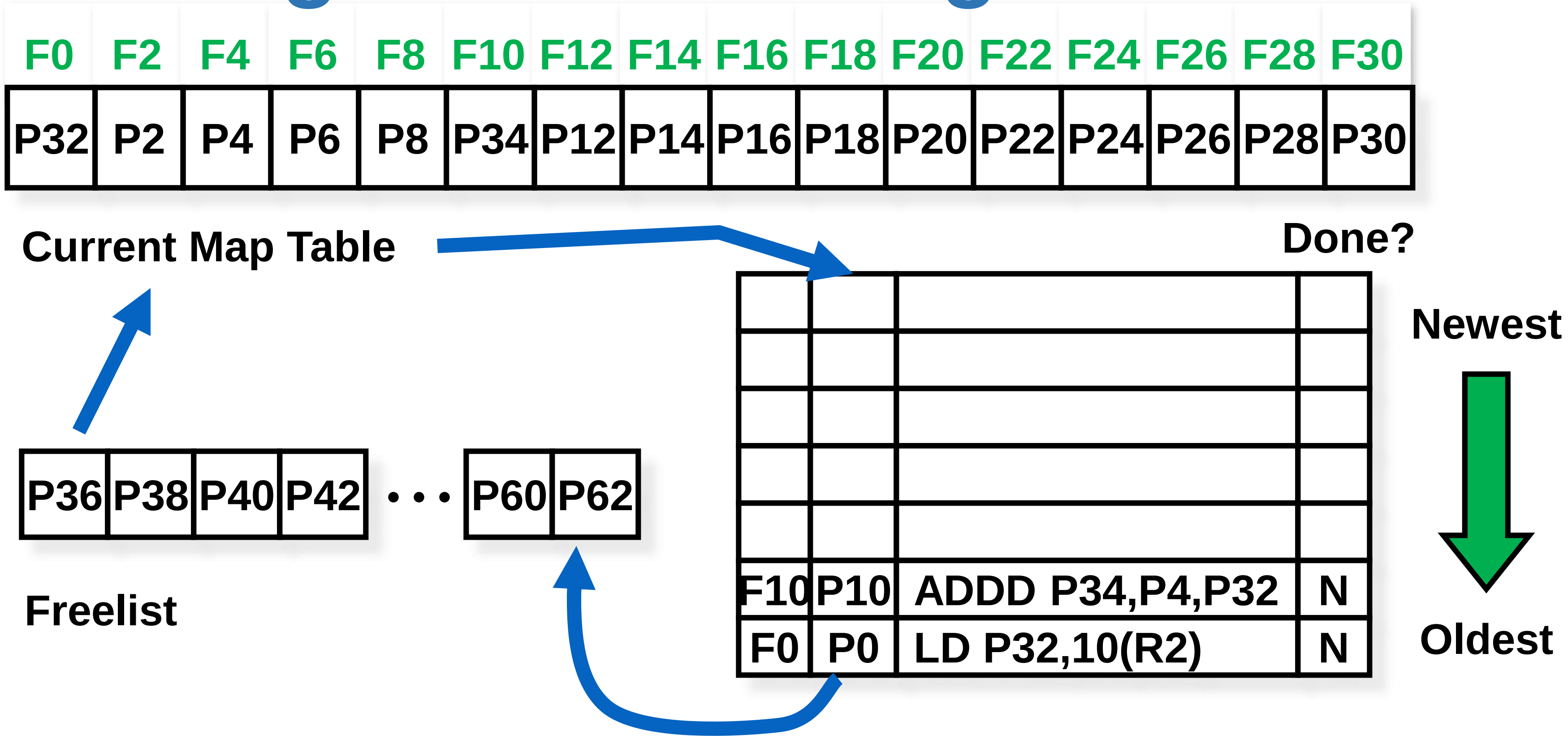
- Hardware equivalent of static, single-assignment (SSA) compiler form
- Physical register file bigger than ISA register file (e.g. 32 Phys regs και 16 ISA regs)
- Upon issue, every instruction that write a register allocates a new physical register from the freelist

Explicit register renaming



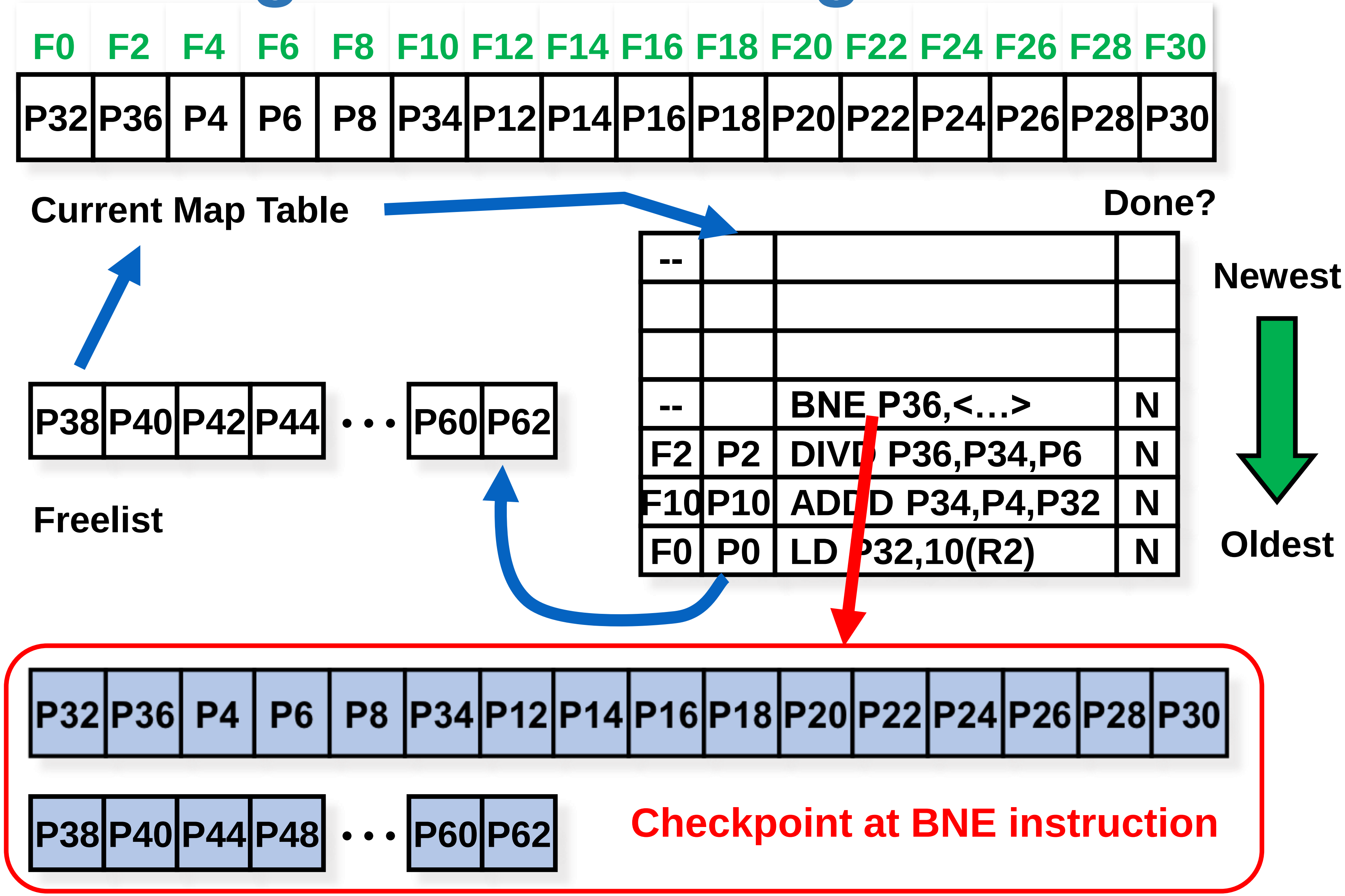
- Note that physical register P0 is “dead” (or not “live”) past the point of this load.
  – When we commit the load, we free up

Explicit register renaming



Issue ADD F10,F4, F0

Explicit register renaming



Explicit register renaming

F0	F2	F4	F6	F8	F10	F12	F14	F16	F18	F20	F22	F24	F26	F28	F30
P40	P36	P38	P6	P8	P34	P12	P14	P16	P18	P20	P22	P24	P26	P28	P30

Current Map Table

			Done?
--		ST 0(R3),P40	Y
F0	P32	ADDD P40,P38,P6	Y
F4	P4	LD P38,0(R3)	Y
--		BNE P36,<…>	N
F2	P2	DIVD P36,P34,P6	N
F10	P10	ADDD P34,P4,P32	Y
F0	P0	LD P32,10(R2)	Y

Newest → Oldest

P42	P44	P48	P50	• • •	P0	P10

Freelist

P32	P36	P4	P6	P8	P34	P12	P14	P16	P18	P20	P22	P24	P26	P28	P30

P38	P40	P44	P48	• • •	P60	P62

Checkpoint at BNE instruction

Explicit register renaming

Speculation error fixed by restoring map table and freelist