

Computer Science 246

Computer Architecture

Spring 2010

Harvard University

Instructor: Prof. David Brooks

dbrooks@eecs.harvard.edu

Dynamic Branch Prediction, Speculation,
and Multiple Issue

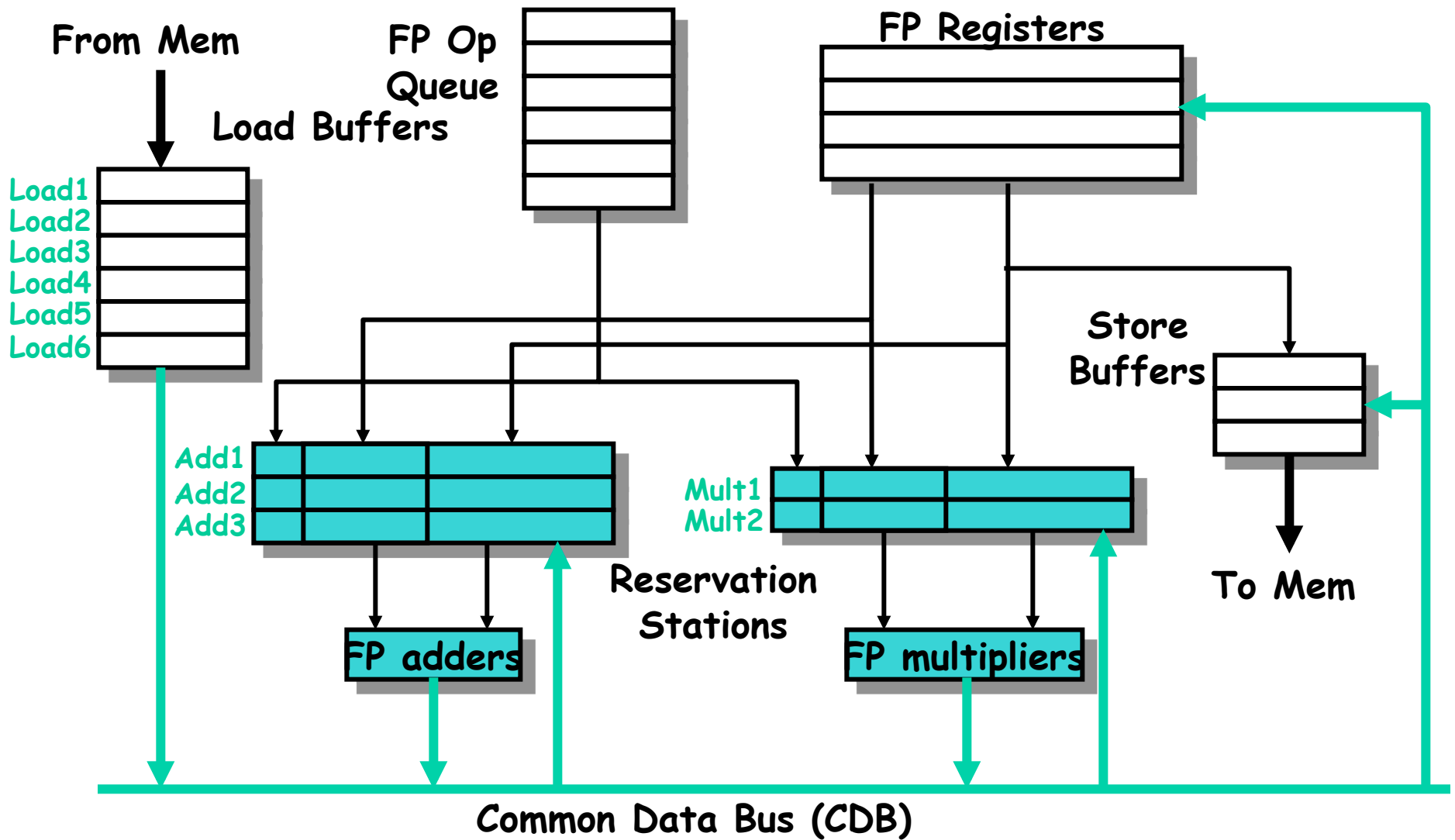
Lecture Outline

- Tomasulo's Algorithm Review (3.1-3.3)
- Pointer-Based Renaming (MIPS R10000)
- Dynamic Branch Prediction (3.4)
- Other Front-end Optimizations (3.5)
 - Branch Target Buffers/Return Address Stack

Tomasulo Review

- Reservation Stations
 - Distribute RAW hazard detection
 - Renaming eliminates WAW hazards
 - Buffering values in Reservation Stations removes WARs
 - Tag match in CDB requires many associative compares
- Common Data Bus
 - Achilles heal of Tomasulo
 - Multiple writebacks (multiple CDBs) expensive
- Load/Store reordering
 - Load address compared with store address in store buffer

Tomasulo Organization



Tomasulo Review

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
LD F0, 0(R1)	Iss	M1	M2	M3	M4	M5	M6	M7	M8	Wb										
MUL F4, F0, F2		Iss	Iss	Iss	Iss	Iss	Iss	Iss	Iss	Iss	Ex	Ex	Ex	Ex	Wb					
SD 0(R1), F0			Iss	Iss	Iss	Iss	Iss	Iss	Iss	Iss	Iss	Iss	Iss	Iss	Iss	M1	M2	M3	Wb	
SUBI R1, R1, 8				Iss	Ex	Wb														
BNEZ R1, Loop					Iss	Ex	Wb													
LD F0, 0(R1)						Iss	Iss	Iss	Iss	M	Wb									
MUL F4, F0, F2							Iss	Iss	Iss	Iss	Iss	Ex	Ex	Ex	Ex	Wb				
SD 0(R1), F0								Iss	Iss	Iss	Iss	Iss	Iss	Iss	Iss	Iss	M1	M2	M3	Wb
SUBI R1, R1, 8									Iss	Ex	Wb									
BNEZ R1, Loop										Iss	Ex	Wb								
LD F0, 0(R1)											Iss	M1	M2	M3	M4	M5	M6	M7	M8	Wb
MUL F4, F0, F2																Iss	Iss	Iss	Iss	Iss
SD 0(R1), F0																	Iss	Iss	Iss	Iss

Register Renaming: Pointer-Based

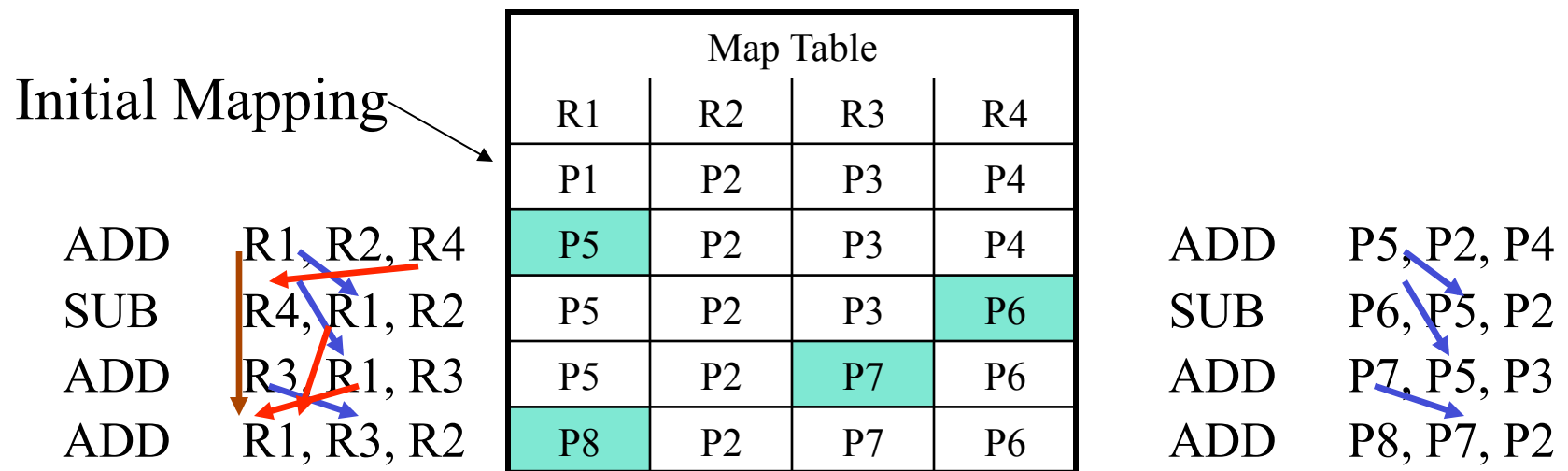
- MIPS R10K, Alpha 21264, Pentium 4, POWER4
- Mapper/Map Table: Hardware to hold these mappings
 - Register Writes: Allocate new location, note mapping in table
 - Register Reads: Look in map table, find location of most recent write
- Deallocate mappings when done

Register Renaming: Example

- Mapper/Map Table: Hardware to hold these mappings
 - Register Writes: Allocate new location, note mapping in table
 - Register Reads: Look in map table, find location of most recent write
- Deallocate mappings when done
- Assume
 - 4 Architected/Logical Registers (F1,F2,F3,F4) “names”
 - 8 Physical/Rename Registers (P1—P8) “locations”
- Code – Lots of Potential WAR/WAW, also RAWs

ADD R1, R2, R4
SUB R4, R1, R2
ADD R3, R1, R3
ADD R1, R3, R2

Register Renaming: Example



Control Hazards

- Key to performance in current microprocessors
- Almost every design decision changes if we assume “perfect” rather than realistic branch prediction

Strategies to reduce control hazards

- Compiler techniques reduce branch frequency
- Hardwired strategies for responding to branches – “assume not taken”
- Delayed branches
- Nullifying branches
- Compiler hints to suggest likely outcomes
- Dynamic hardware branch prediction

Branch prediction methods

- When is information about branches gathered/applied?
 - When the machine is designed
 - When the program is compiled (“compile-time”) (ch.4)
 - When a “training run” of the program is executed (“profile-based”)
 - As the program is executing (“dynamic”)

Why predict? Speculative Execution

- Execute *beyond* branch boundaries before the branch is resolved
- Correct Speculation
 - Avoid stall, result is computed early, performance++
- Incorrect Speculation
 - Abort/squash incorrect instructions, complexity+
 - Undo any incorrect state changes, complexity++
- Performance gain is weighed vs. penalty
- Speculation accuracy = branch prediction accuracy

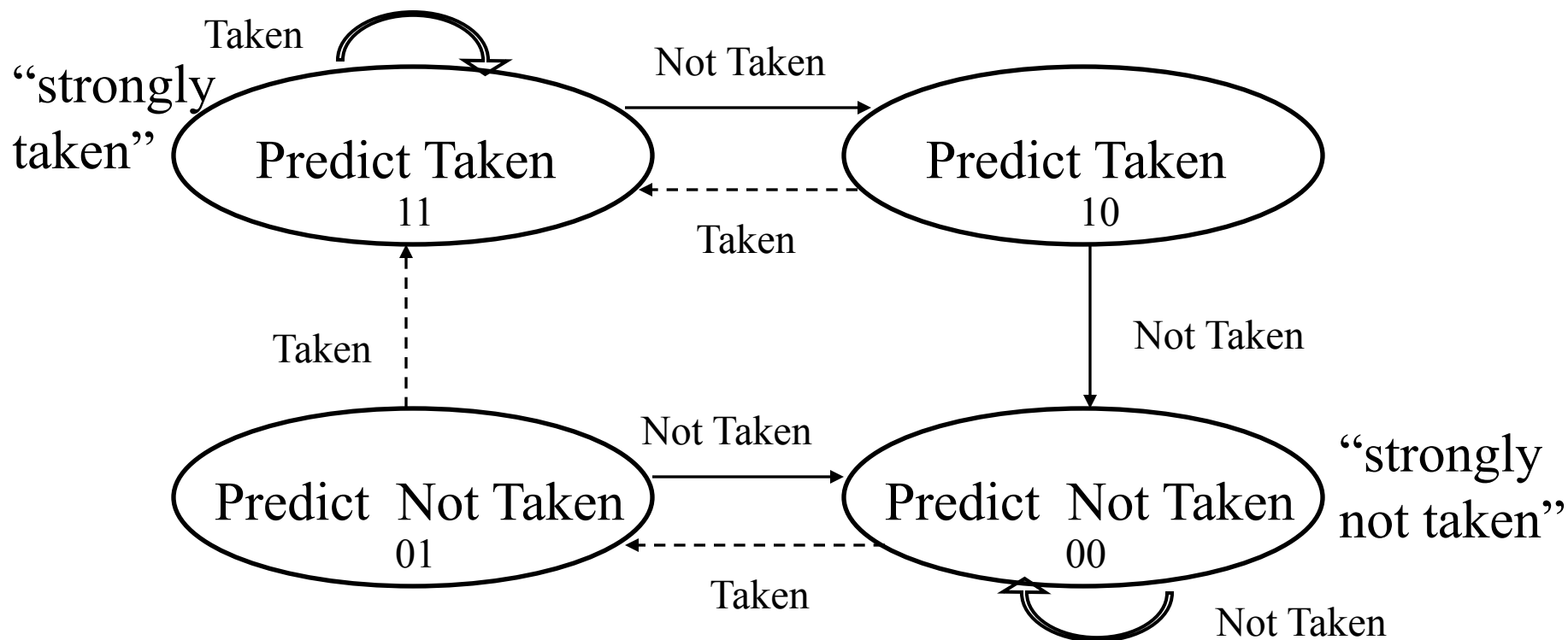
Dynamic Hardware Branch Prediction

- Branch behavior is monitored during program execution
 - History data can influence prediction of future executions of the branch instruction
- Branches instruction execution has two tasks/predictions
 - Condition evaluation (taken or not-taken)
 - Target address calculation (where to go when taken)
- Target prediction also applies to unconditional branches
- Branch Direction Prediction: 3 levels of complexity
 - Branch history tables, Two-level tables, hybrid predictors

Branch Direction Prediction

- Basic idea: Hope that future behavior of the branch is correlated to past behavior
 - Loops
 - Error-checking conditionals
- For a single branch PC
 - Simplest possible idea: Keep 1 bit around to indicate taken or not-taken
 - 2nd simplest idea: Keep 2 bits around, saturating counter

Two-bit Saturating Counters



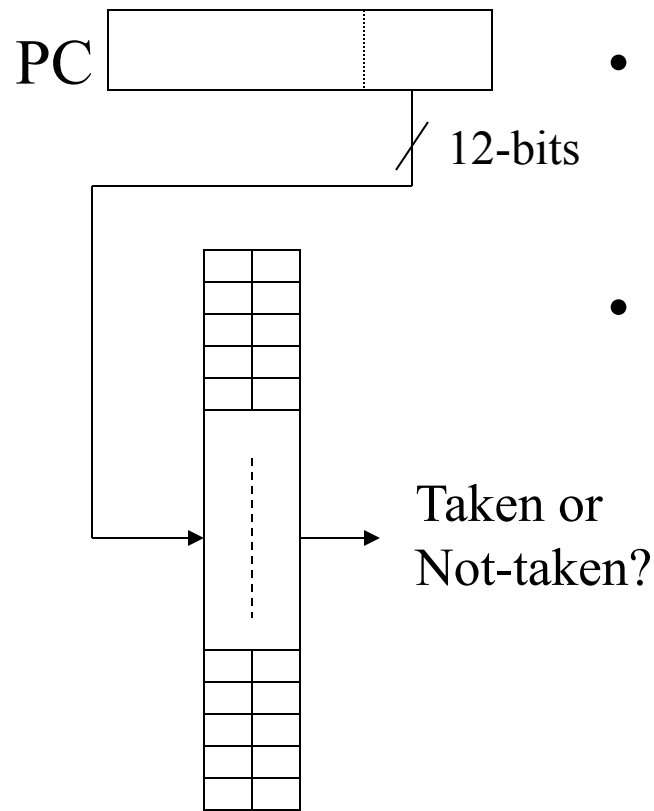
- 2-bit FSMs mean prediction must miss twice before change
- N-bit predictors are possible, but after 2-bits not much benefit

Example: Two-bit vs. 1-bit Branch Prediction

Branch Outcome	T	T	T	N	T	T	T	N	T	T	T	N	% predict rate
1-bit Prediction	N	T	T	T	N	T	T	T	N	T	T	T	
1-bit Mis-Predict?	Y			Y	Y			Y	Y			Y	~50%
2-bit Prediction	n	T	T	t	T	T	T	t	T	T	T	t	
2-bit Mis-Predict?	Y			Y				Y				Y	~75%

- 2-bit “hysteresis” helps

Branch Prediction Buffer (branch history table, BHT)



$2^{12} = 4K$ Entries

- Small memory indexed with low bits of the branch instruction's address
 - Why the low bits?
- Implementation
 - Separate memory accessed during IF phase
 - 2-bits attached to each block in the Instruction Cache
 - Caveats: Cannot separately size I-Cache and BHT
 - What about multiple branches in a cache line?
 - Does this help our simple 5-stage pipeline?

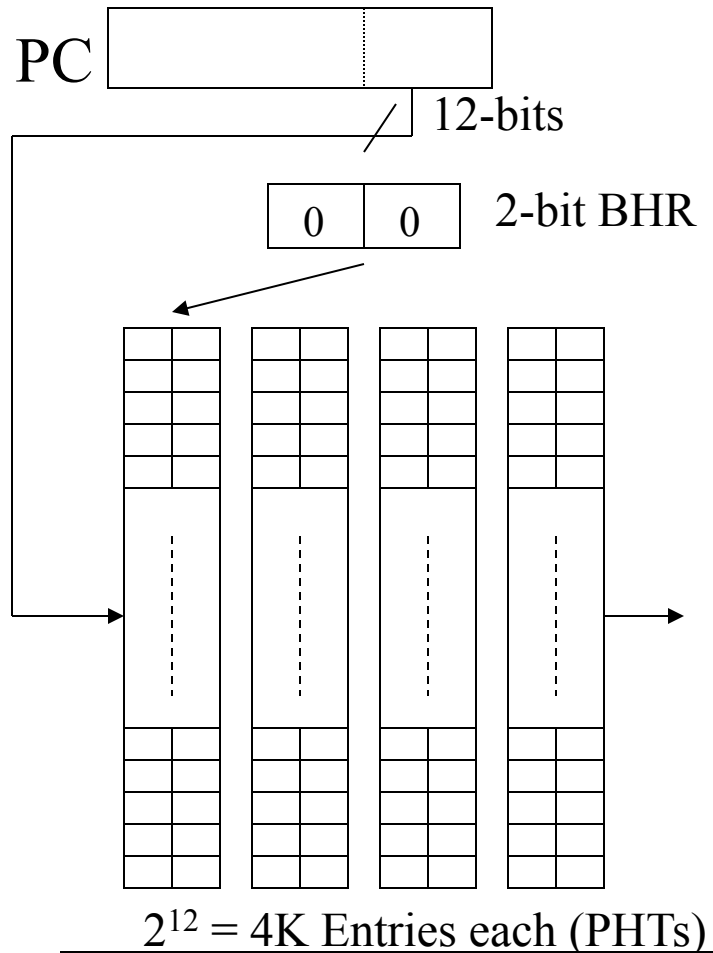
Correlating Predictors

- 2-bit scheme only looks at branch's *own* history to predict its behavior
- What if we use other branches to predict it as well?

```
if (aa==2) aa=0;    // Branch #1
if (bb==2) bb=0;    // Branch #2
if (aa!=bb) { .. } // Branch #3
```

- Clearly branch #3 depends on outcome of #1 and #2
- Prediction must be a function of own branch as well as recent outcomes of other branches

Two-level Adaptive Branch Prediction (Correlating Predictor)



- Two-level BP requires two main components
 - Branch history register (BHR): recent outcomes of branches (last **k** branches encountered)
 - Pattern History Table (PHT): branch behavior for last **s** occurrences of the specific pattern of these **k** branches
 - In effect, we concatenate BHR with Branch PC bits
 - Can also XOR (GSHARE), etc

Branch History Register

- Simple shift register
 - Shift in branch outcomes as they occur
 - 1 \Rightarrow branch was taken
 - 0 \Rightarrow branch was not-taken
 - k-bit BHR $\Rightarrow 2^k$ patterns
 - Use these patterns to address into the Pattern History Table

Pattern History Table

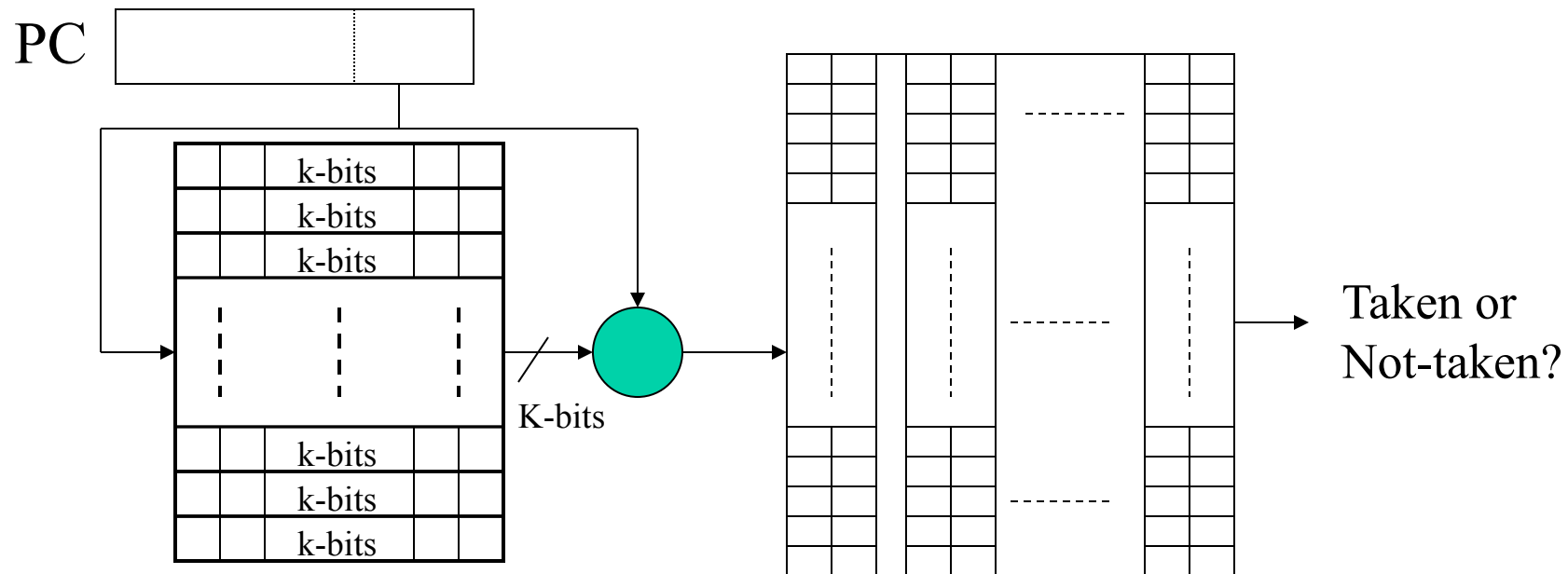
- Has 2^k entries
- Usually uses a 2-bit counter for the prediction
- Each entry summarizes branch results for the last s times that BHR pattern was seen
 - Not a shift register, usually a FSM
- BHR is used to address the PHT along with PC bits

Variations on 2-Level BP

- Variations depend on
 - How many branches share a BHR
 - How many branches share a PHT
- 3 possibilities for each: global, per-address, per-set

2-level Branch History

- Global history -- 1 Branch History Register (BHR)
- Per-address/set history
 - Per-Address/set Branch History Table holds many BHRs



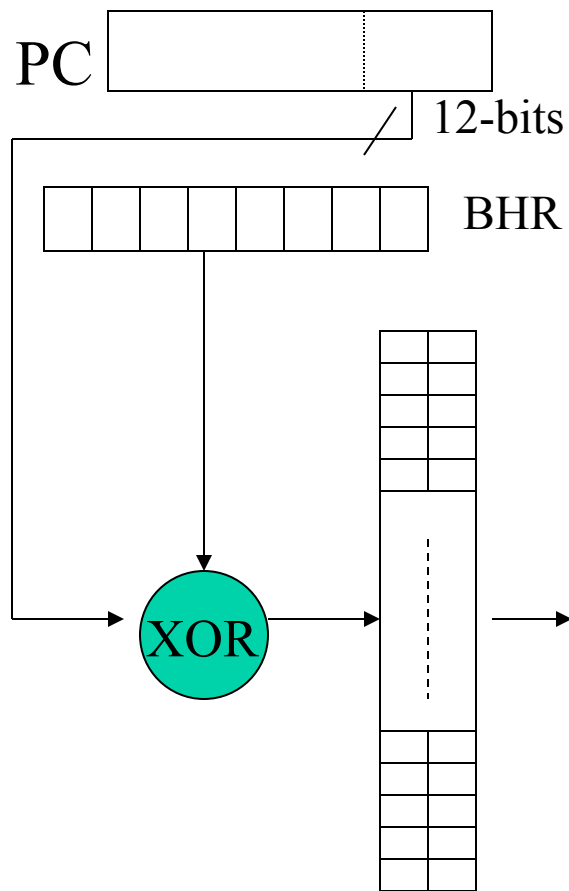
Hardware Costs of 2-level predictions

- (m,n) predictor → m-bits of global history, n-bit predictor
- $2^m * n * \text{Number of prediction entries}$
- Say you have m-bits of history (m=2)
- n-bits of predictor per entries (n=2)

(2,2) predictor with 1K prediction entries

$$2^2 * 2 * 1024 = 8\text{K-bits}$$

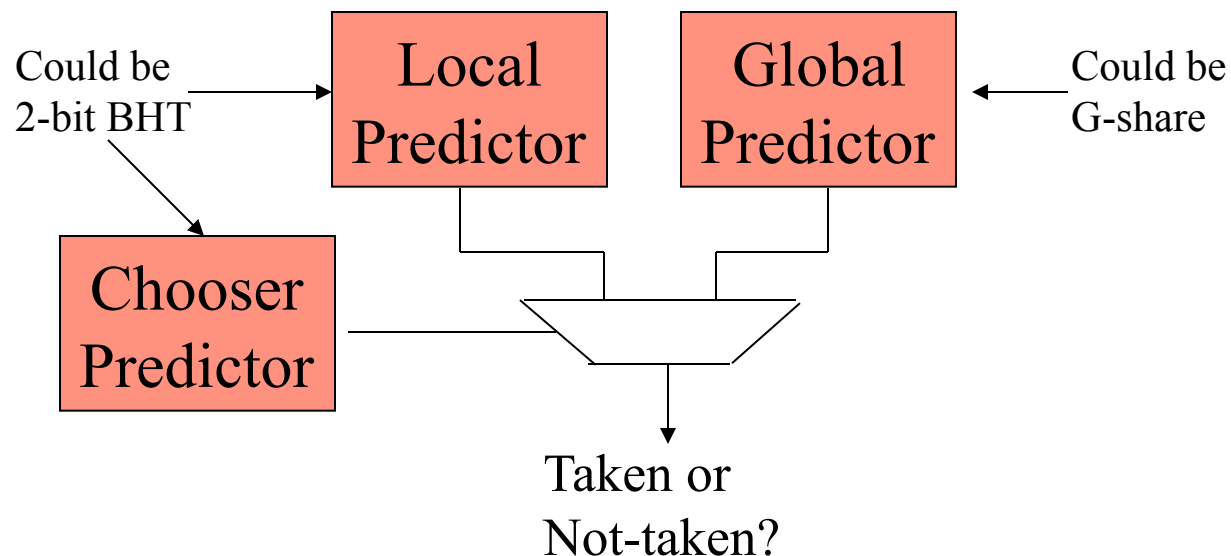
Variations on the basics -- GSHARE



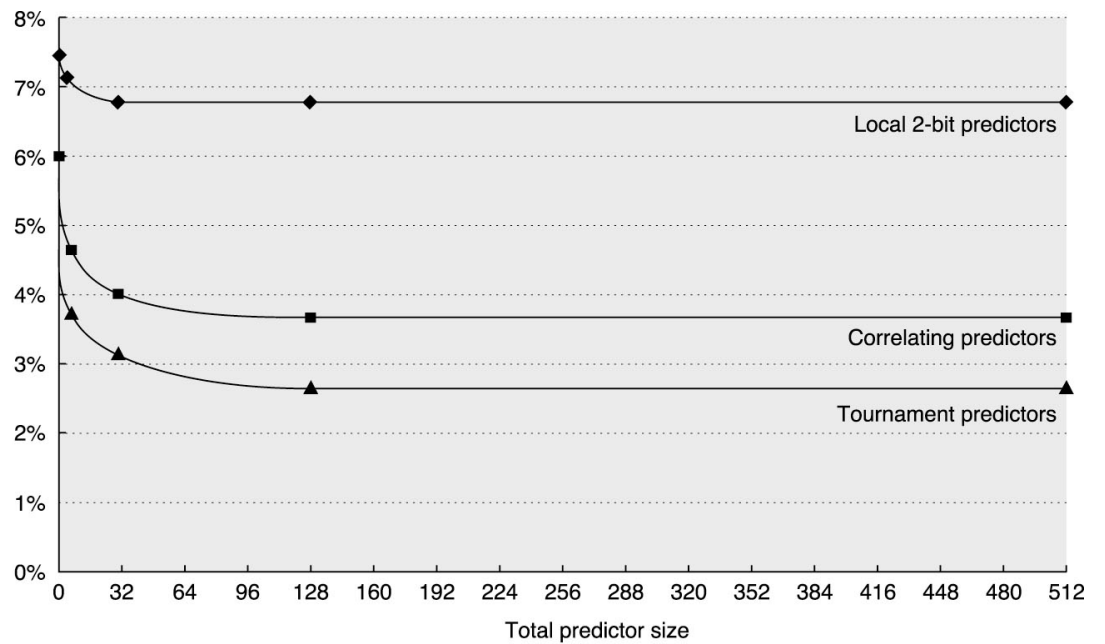
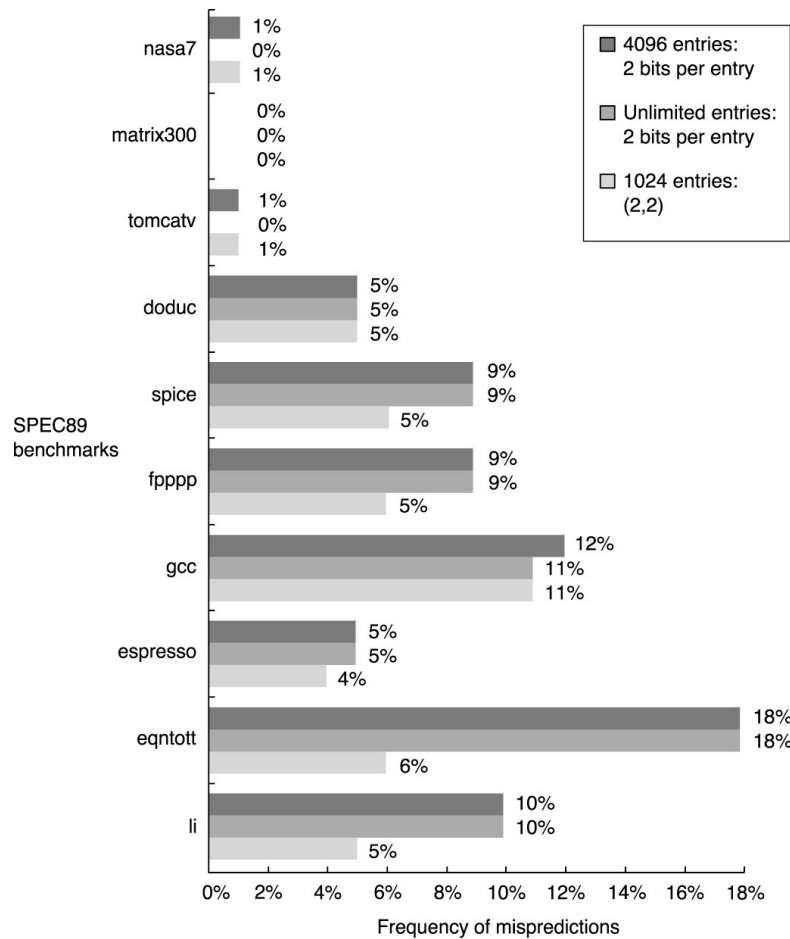
- Gshare a variant on GAg
- Don't use BHR directly to address PHT
- Instead, XOR bits of BHR with bits of PC (branch address) and use that to index PHT
- Tries to separate out the behaviors/predictions associated with different branches, without extra hardware of PA and SA schemes

Hybrid Branch Predictors

- Tournament predictors: Adaptively combine local and global predictors
- Different schemes work better for different branches



Branch Predictor Performance



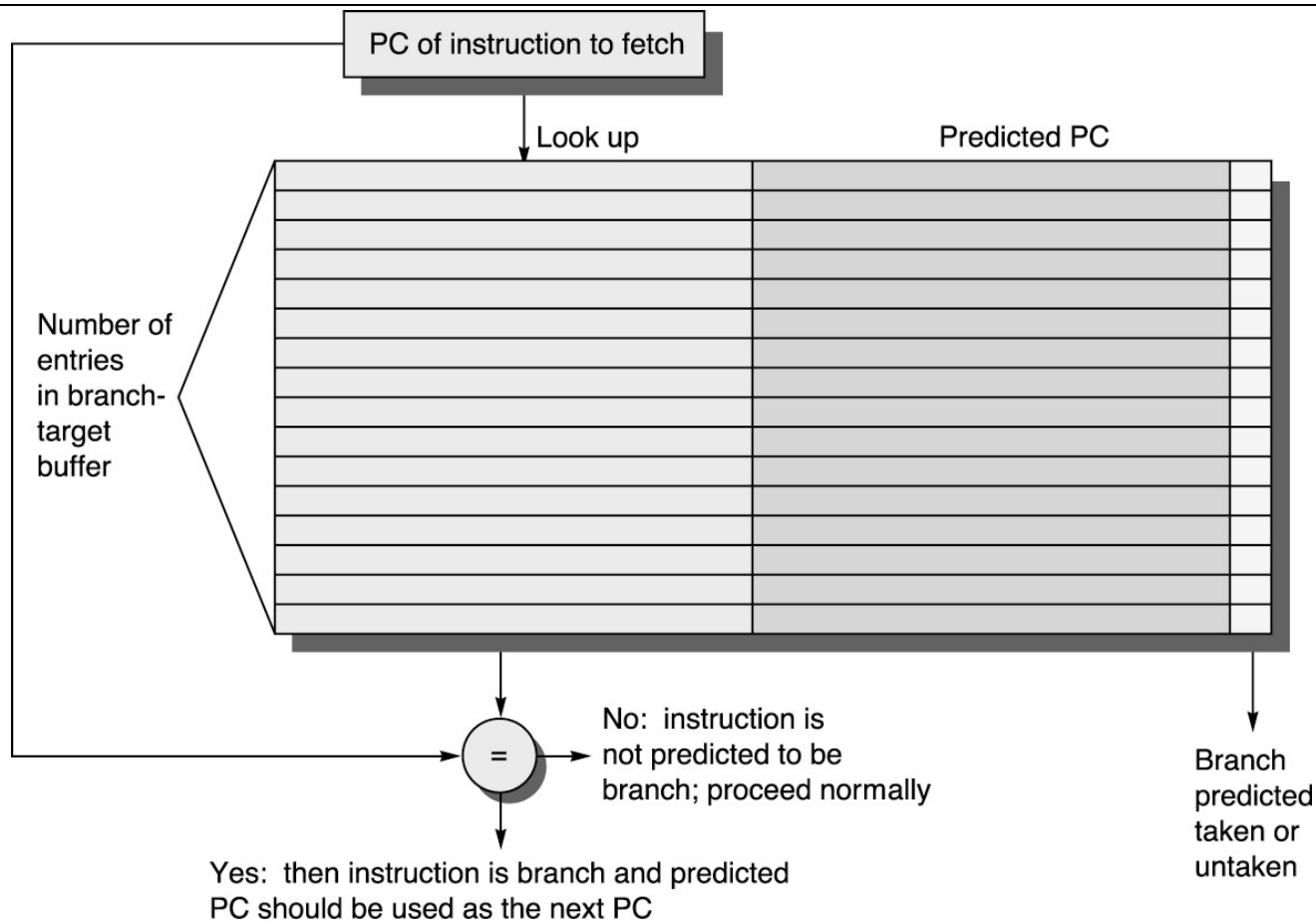
Branch Target Prediction

- So far we have only talked about predicting *direction*
- We still need to predict the *address*
 - Branch Target Buffer (BTB)
 - Useful for conditional/unconditional branches
 - Return Address Stack (RAS)
 - Useful for procedure returns

Branch Target Buffer

- Simple pipeline resolves stages in ID
 - We'd really like to know by the end of IF so we can proceed without a bubble
- Idea:
 - As part of IF use the instruction address (every instruction) to do a lookup in the BTB
 - For N recently executed branches, hold the predicted PC value (may also hold additional prediction bits)
 - If instruction is not a branch, don't add to BTB
 - If BTB fails revert to earlier method
 - Either instruction is not a branch
 - Or, there is no predictor entry for that branch
 - Many more bits per entry than BHT

Branch Target Buffer



Branch Target Cache

- Similar to BTB, but we also want to know the target instruction!
 - Prediction returns not just the direction address, but also the instruction stored there
 - Allows zero-cycle branches (branch-folding)
 - Send target-instruction to ID rather than branch
 - Branch is not sent into pipe

Return Address Stack

- Included in many recent processors
 - Alpha 21264 => 12 entry RAS
- Procedure returns account for ~85% of indirect jumps
- Like a hardware stack, LIFO
 - Procedure Call => Push Return PC onto stack
 - Procedure Return => Prediction off of top of stack, Pop it
- RAS tends to work quite well since call depths are typically not large

Return Address Stack

- Say `foo()` is called from many different locations in a program
- It will then return to many different locations!
- RAS can predict which location to return to because it stores the caller PC
- This is faster than having to load up indirect jumps (`jump r31`)
- If the call-depth doesn't exceed the size of the RAS, this prediction will always be correct

Putting things together

- Talked about these things independently...
- Instruction Fetch
 - Branch Prediction (fill scheduler with instruction + multiple instruction per cycle)
- Scheduling/Hazard elimination
 - Dynamic Scheduling with Tomasulo (RAW Hazards)
 - Register Renaming (WAR and WAW Hazards)
- Multiple functional units, register file ports
 - Potentially can reduce $CPI < 1$
- Speculative Execution
- Precise Interrupts
- Memory systems (later this semester)

Focus on Speculation/Interrupts

- Precise Interrupts
 - All instructions before interrupt must complete
 - All instructions after interrupt must seem to never start
- Speculation (similar problem!)
 - If branch prediction is wrong, could update state incorrectly leading to wrong program behavior
- Out-of-Order *completion*
 - Post-interrupt/mispredict writebacks change state
 - Does Out-of-Order scheduling require this?

Solving both problems with one solution

- Need the ability to squash/restart *any* instruction
 - Gives us precise state
 - Need for memory ops (page faults, etc)
 - Need for FP ops (divide by 0)
 - Gives us ability to recover mis-speculations
 - Need for branches
- Providing precise state solves both these problems

How to get precise state?

- Imprecise state
 - As we've said this is a bad idea
 - For speculation it is unacceptable
- Force in-order completion at WB (stall when necessary)
- Precise state in software: save recovery info for traps
 - Traps on all faulting memory, FP, and mis-predicted branch ops?
- Precise state in hardware: save recovery info online

Solution: Writeback and Commit

- Allow out of order issue/writeback
 - Require in-order commit when instruction is no longer speculative
 - Prevent speculative changes from changing state
 - e.g. memory write or register write
- Collect pre-commit instructions
 - in a reorder buffer
 - holds completed but not committed instruction
 - Effectively contains a set of virtual registers
 - similar to a reservation station
 - and becomes a bypass (forwarding) source

Four 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”)

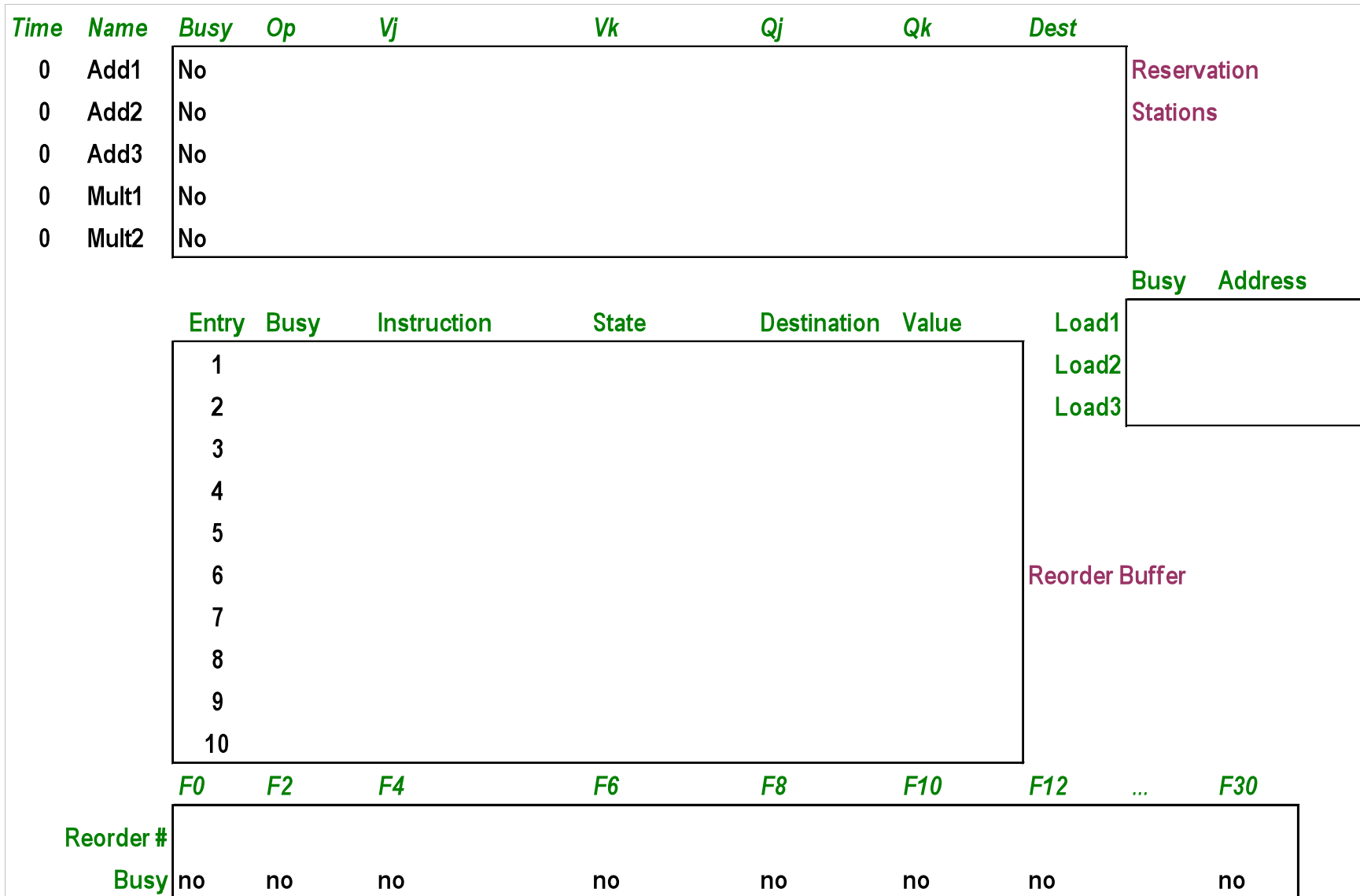
3. Write result—finish execution (WB)

Write on Common Data Bus to all awaiting FUs & reorder buffer; mark reservation station available. (tags are now ROB #s not RS #s)

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 - Cycle 0



Tomasulo With Reorder Buffer - Cycle 1

Time	Name	Busy	Op	Vj	Vk	Qj	Qk	Dest
0	Add1	No						
0	Add2	No						
0	Add3	No						
0	Mult1	No						
0	Mult2	No						

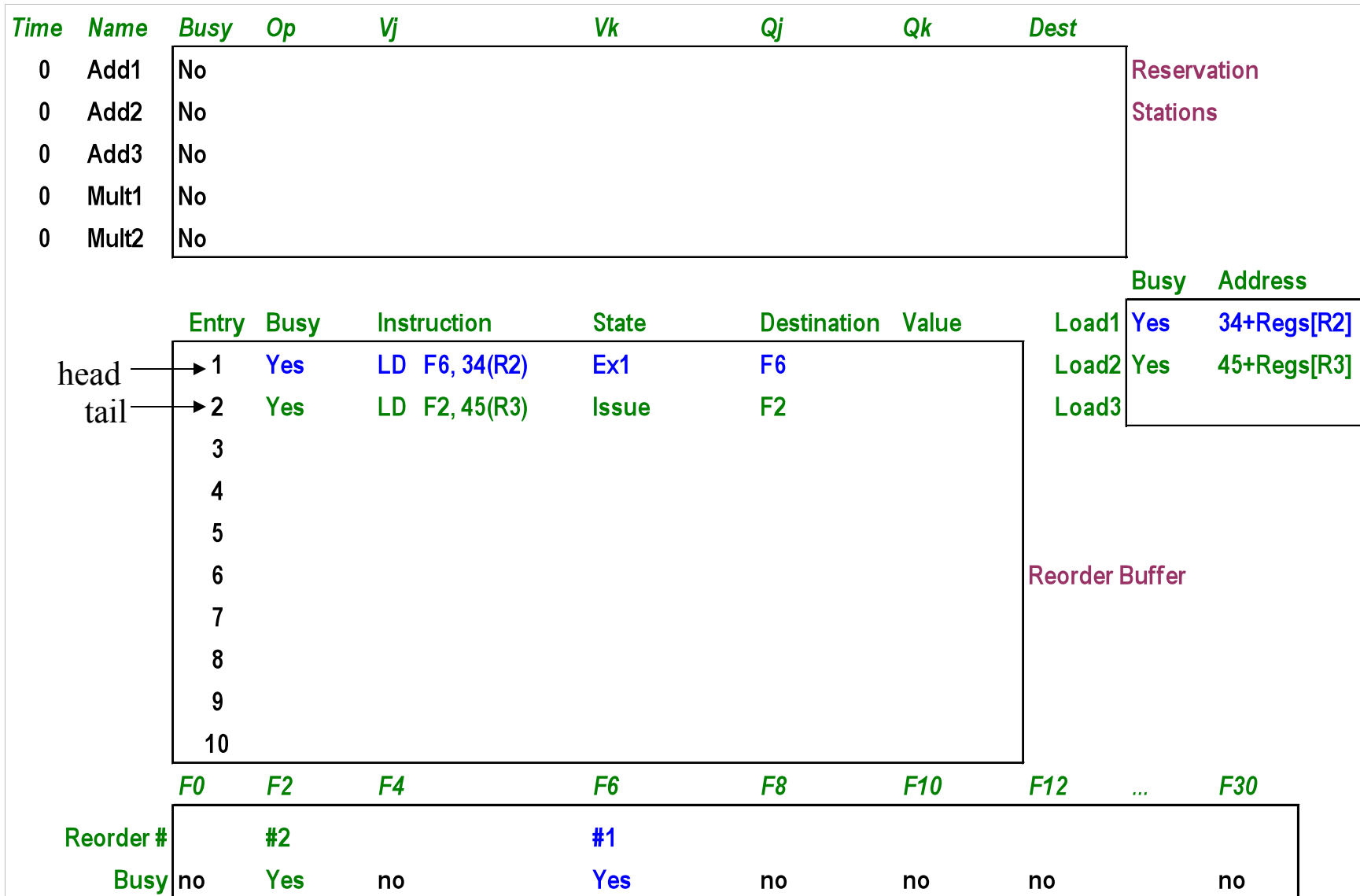
Reservation Stations

Entry	Busy	Instruction	State	Destination	Value	Load1	Load2	Load3
1	Yes	LD F6, 34(R2)	Issue	F6		Yes		34+Regs[R2]
2								
3								
4								
5								
6								
7								
8								
9								
10								

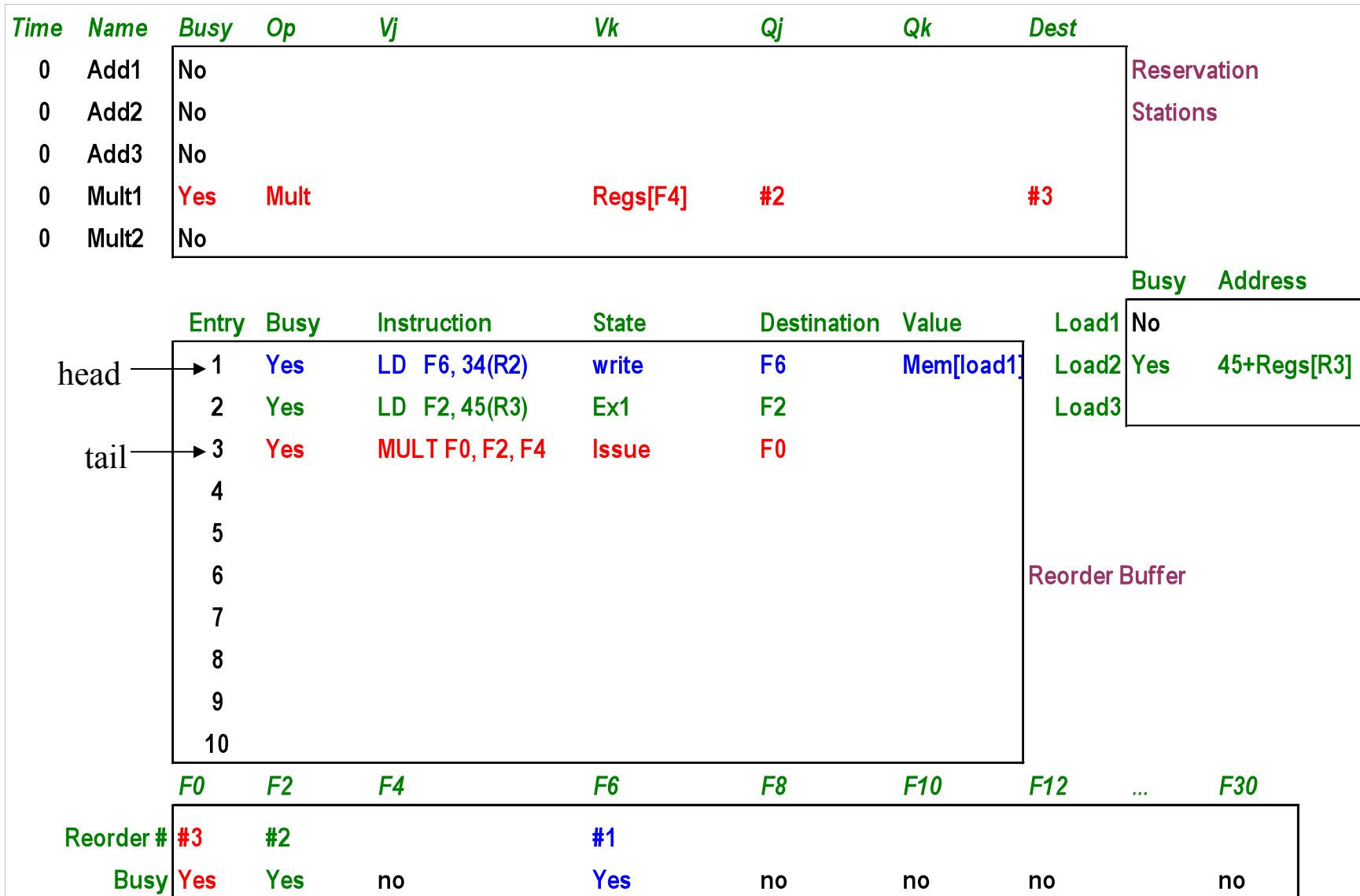
Reorder Buffer

Reorder #	F0	F2	F4	F6	F8	F10	F12	...	F30
				#1					
Busy	no	no	no	Yes	no	no	no		no

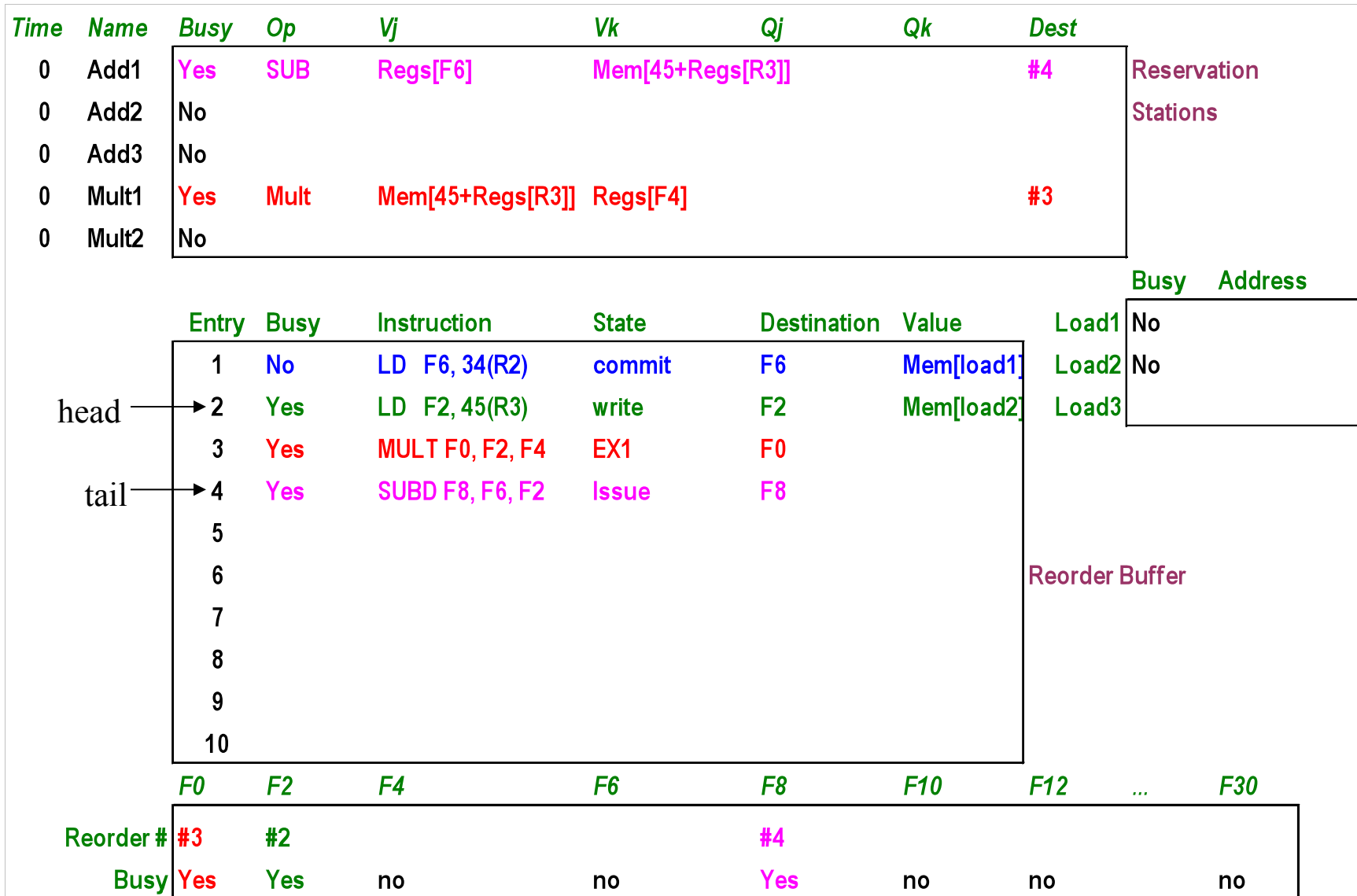
Tomasulo With Reorder Buffer - Cycle 2



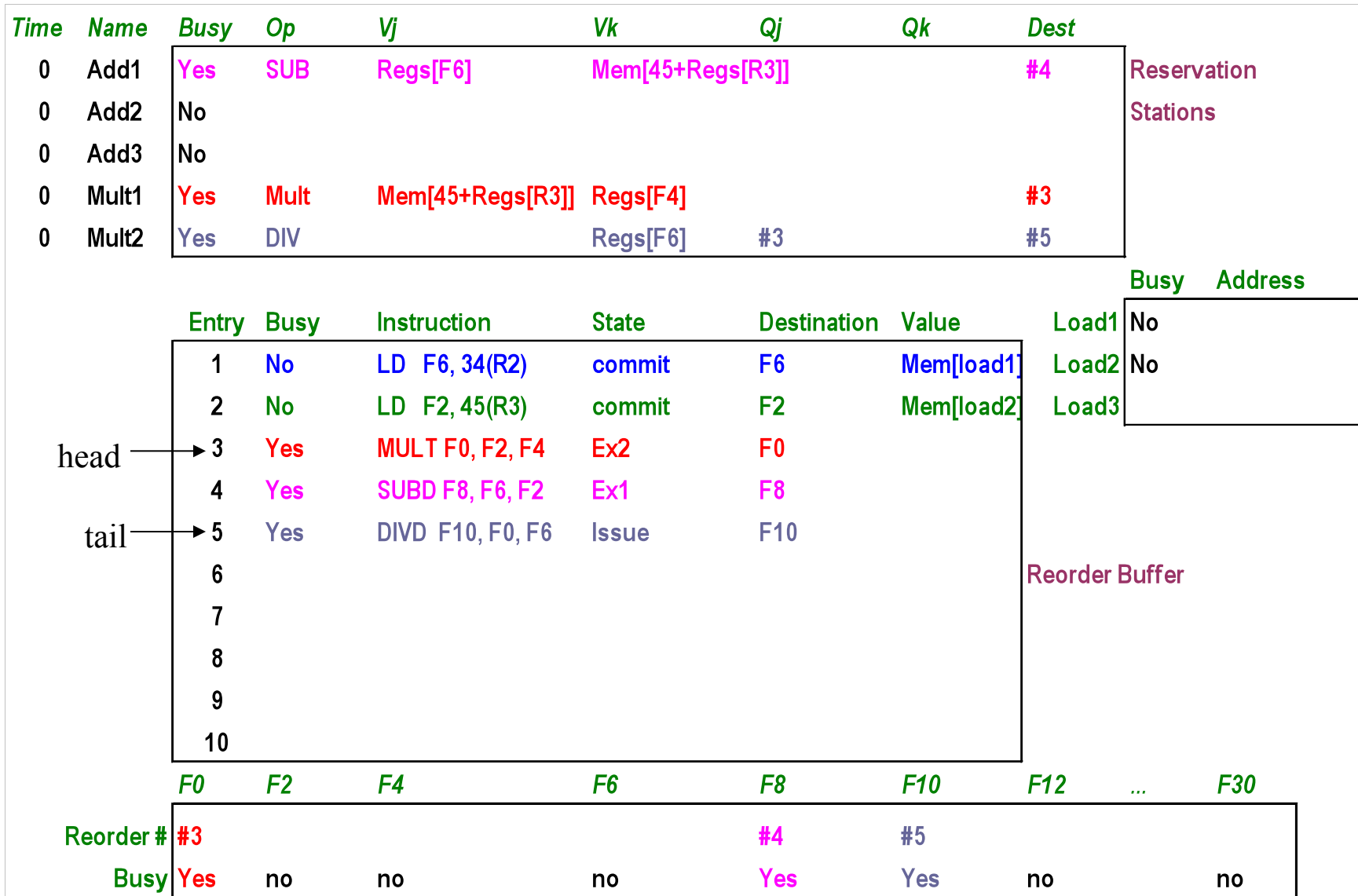
Tomasulo With Reorder Buffer - Cycle 3



Tomasulo With Reorder Buffer - Cycle 4



Tomasulo With Reorder Buffer - Cycle 5



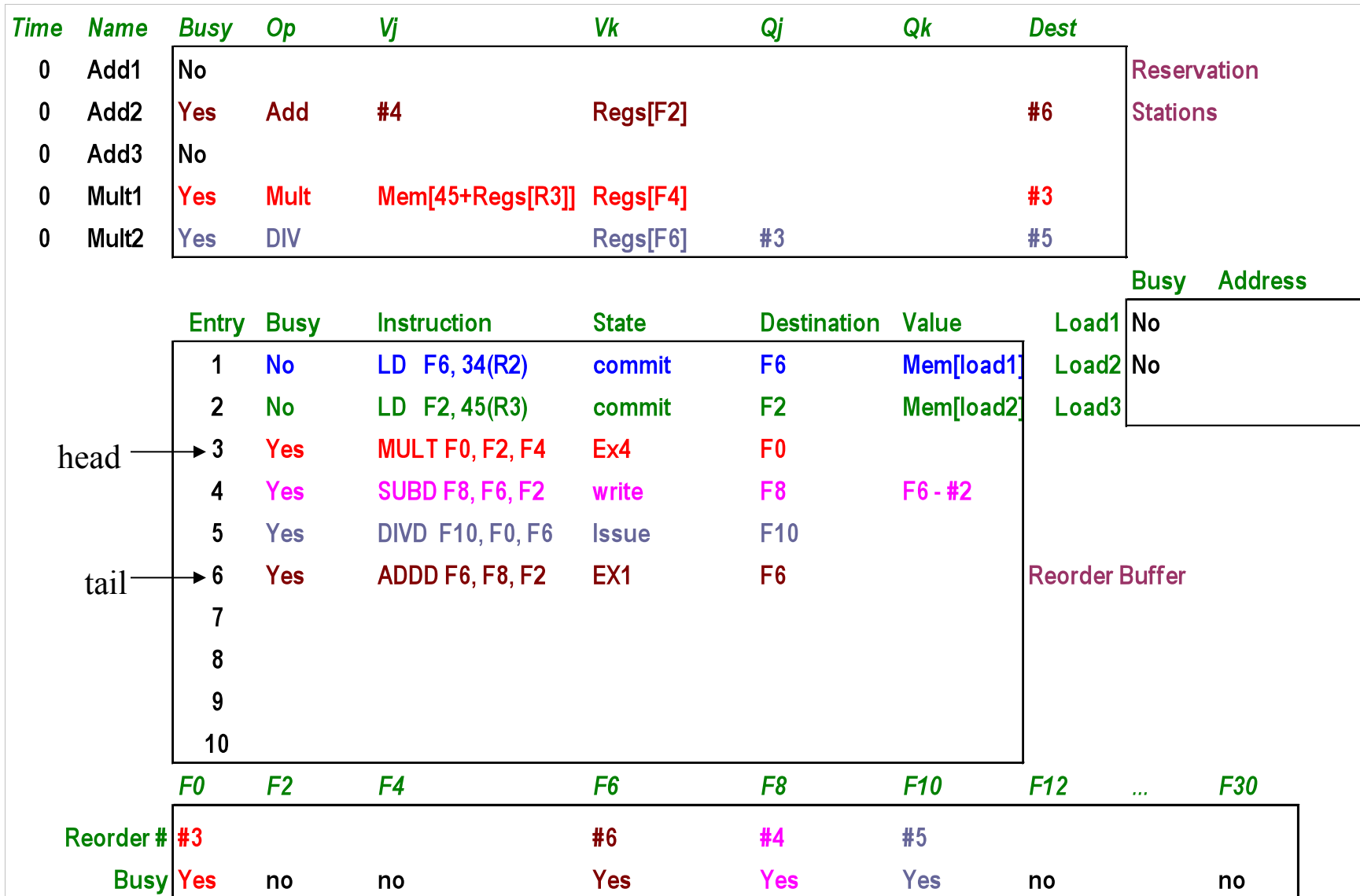
Tomasulo With Reorder Buffer - Cycle 6

Time	Name	Busy	Op	Vj	Vk	Qj	Qk	Dest	
0	Add1	Yes	SUB	Regs[F6]	Mem[45+Regs[R3]]			#4	Reservation Stations
0	Add2	Yes	Add		Regs[F2]	#4		#6	
0	Add3	No							
0	Mult1	Yes	Mult	Mem[45+Regs[R3]]	Regs[F4]			#3	
0	Mult2	Yes	DIV		Regs[F6]	#3		#5	

Entry	Busy	Instruction	State	Destination	Value	Load1	Load2	Load3	Busy	Address
1	No	LD F6, 34(R2)	commit	F6	Mem[load1]	No	No	No		
2	No	LD F2, 45(R3)	commit	F2	Mem[load2]	No	No	No		
3	Yes	MULT F0, F2, F4	Ex3	F0						
4	Yes	SUBD F8, F6, F2	Ex2	F8						
5	Yes	DIVD F10, F0, F6	Issue	F10						
6	Yes	ADDD F6, F8, F2	Issue	F6						
7										
8										
9										
10										

Reorder #	F0	F2	F4	F6	F8	F10	F12	...	F30
#3				#6	#4	#5			
Busy	Yes	no	no	Yes	Yes	Yes	no		no

Tomasulo With Reorder Buffer - Cycle 7



Tomasulo With Reorder Buffer - Cycle 8

Time	Name	Busy	Op	Vj	Vk	Qj	Qk	Dest	
0	Add1	No							Reservation Stations
0	Add2	Yes	Add	#4	Regs[F2]			#6	
0	Add3	No							
0	Mult1	Yes	Mult	Mem[45+Regs[R3]]	Regs[F4]			#3	
0	Mult2	Yes	DIV		Regs[F6]	#3		#5	

	Entry	Busy	Instruction	State	Destination	Value	Load1	Load2	Load3	Busy	Address
	1	No	LD F6, 34(R2)	commit	F6	Mem[load1]	No	No			
	2	No	LD F2, 45(R3)	commit	F2	Mem[load2]	No	No			
head →	3	Yes	MULT F0, F2, F4	Ex5	F0						
	4	Yes	SUBD F8, F6, F2	write	F8	F6 - #2					
	5	Yes	DIVD F10, F0, F6	Issue	F10						
tail →	6	Yes	ADDD F6, F8, F2	Ex2	F6						Reorder Buffer
	7										
	8										
	9										
	10										

	F0	F2	F4	F6	F8	F10	F12	...	F30
Reorder #	#3			#6	#4	#5			
Busy	Yes	no	no	Yes	Yes	Yes	no		no

Tomasulo With Reorder Buffer - Cycle 9

Time	Name	Busy	Op	Vj	Vk	Qj	Qk	Dest	
0	Add1	No							Reservation Stations
0	Add2	Yes	Add	#4	Regs[F2]			#6	
0	Add3	No							
0	Mult1	Yes	Mult	Mem[45+Regs[R3]]	Regs[F4]			#3	
0	Mult2	Yes	DIV		Regs[F6]	#3		#5	

	Entry	Busy	Instruction	State	Destination	Value	Load1	Load2	Load3	Busy	Address
	1	No	LD F6, 34(R2)	commit	F6	Mem[load1]	No	No	No		
	2	No	LD F2, 45(R3)	commit	F2	Mem[load2]	No	No	No		
head →	3	Yes	MULT F0, F2, F4	Ex6	F0						
	4	Yes	SUBD F8, F6, F2	write	F8	F6 - #2					
	5	Yes	DIVD F10, F0, F6	Issue	F10						
tail →	6	Yes	ADDD F6, F8, F2	write	F6	#4 + F2					Reorder Buffer
	7										
	8										
	9										
	10										

	F0	F2	F4	F6	F8	F10	F12	...	F30
Reorder #	#3			#6	#4	#5			
Busy	Yes	no	no	Yes	Yes	Yes	no		no

Tomasulo With Reorder Buffer - Cycle 10

Time	Name	Busy	Op	Vj	Vk	Qj	Qk	Dest
0	Add1	No						
0	Add2	No						
0	Add3	No						
0	Mult1	Yes	Mult	Mem[45+Regs[R3]]	Regs[F4]			#3
0	Mult2	Yes	DIV		Regs[F6]	#3		#5

Entry	Busy	Instruction	State	Destination	Value	Load1	Load2	Load3	Busy	Address
1	No	LD F6, 34(R2)	commit	F6	Mem[load1]	No	No	No		
2	No	LD F2, 45(R3)	commit	F2	Mem[load2]	No	No	No		
3	Yes	MULT F0, F2, F4	Ex7	F0						
4	Yes	SUBD F8, F6, F2	write	F8	F6 - #2					
5	Yes	DIVD F10, F0, F6	Issue	F10						
6	Yes	ADDD F6, F8, F2	write	F6	#4 + F2					Reorder Buffer
7										
8										
9										
10										

Reorder #	F0	F2	F4	F6	F8	F10	F12	...	F30
#3				#6	#4	#5			
Busy	Yes	no	no	Yes	Yes	Yes	no		no

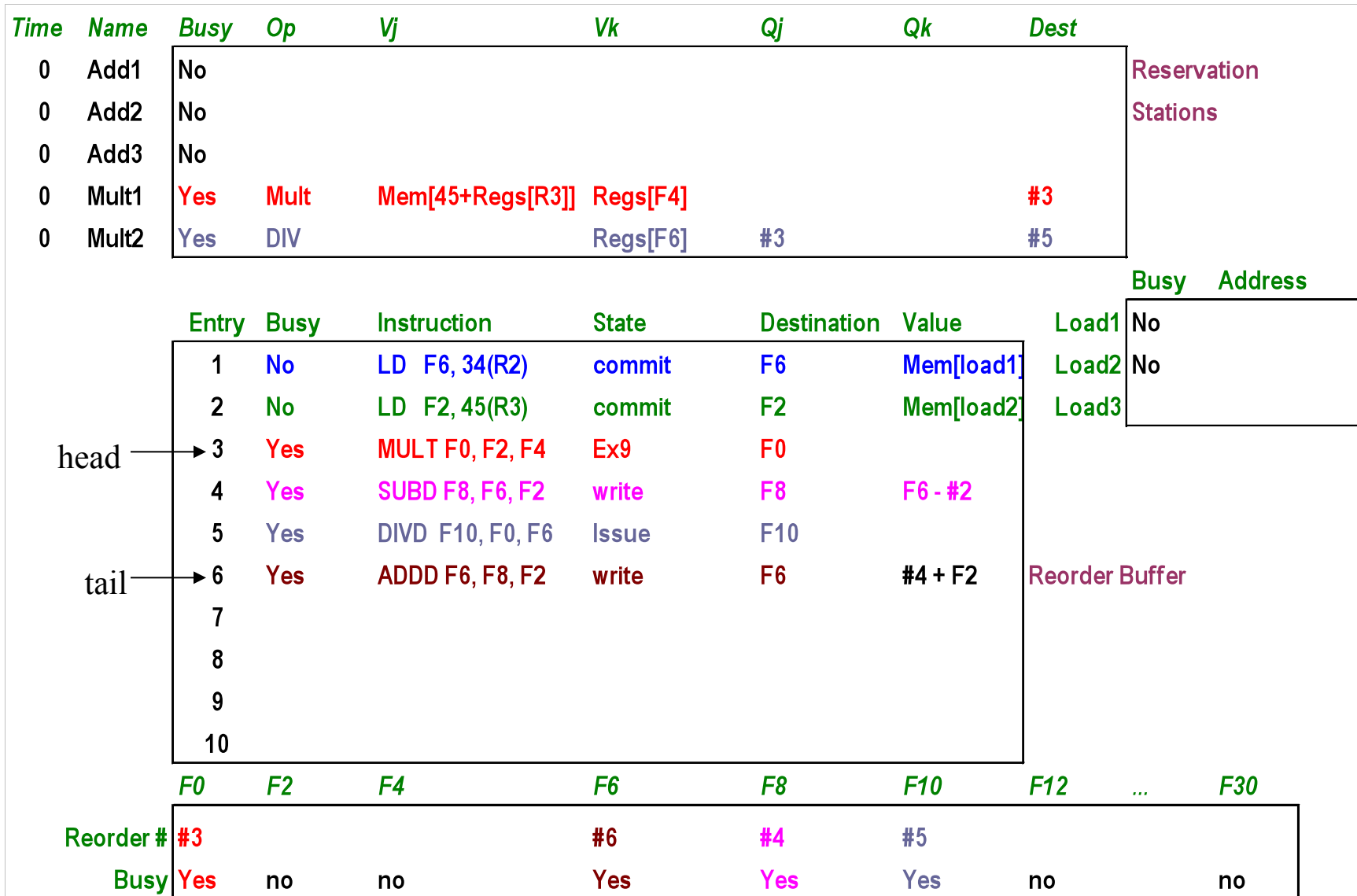
Tomasulo With Reorder Buffer - Cycle 11

Time	Name	Busy	Op	Vj	Vk	Qj	Qk	Dest
0	Add1	No						
0	Add2	No						
0	Add3	No						
0	Mult1	Yes	Mult	Mem[45+Regs[R3]]	Regs[F4]			#3
0	Mult2	Yes	DIV		Regs[F6]	#3		#5

Entry	Busy	Instruction	State	Destination	Value	Load1	Load2	Load3	Busy	Address
1	No	LD F6, 34(R2)	commit	F6	Mem[load1]	No	No	No		
2	No	LD F2, 45(R3)	commit	F2	Mem[load2]	No	No	No		
3	Yes	MULT F0, F2, F4	Ex8	F0						
4	Yes	SUBD F8, F6, F2	write	F8	F6 - #2					
5	Yes	DIVD F10, F0, F6	Issue	F10						
6	Yes	ADDD F6, F8, F2	write	F6	#4 + F2					Reorder Buffer
7										
8										
9										
10										

	F0	F2	F4	F6	F8	F10	F12	...	F30
Reorder #	#3			#6	#4	#5			
Busy	Yes	no	no	Yes	Yes	Yes	no		no

Tomasulo With Reorder Buffer - Cycle 12



Tomasulo With Reorder Buffer - Cycle 13

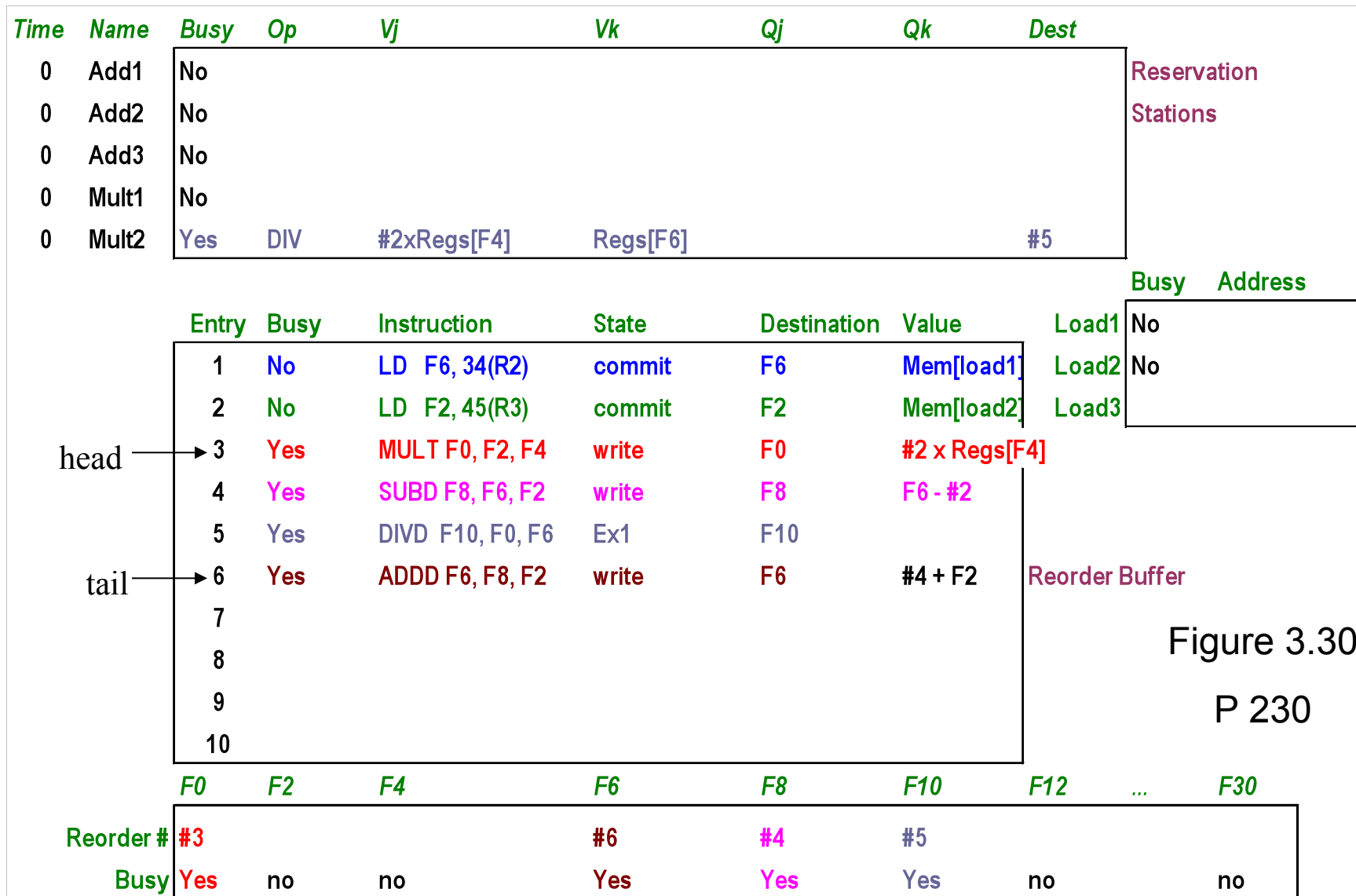


Figure 3.30

P 230

Tomasulo With Reorder Buffer - Cycle 14

Time	Name	Busy	Op	Vj	Vk	Qj	Qk	Dest
0	Add1	No						
0	Add2	No						
0	Add3	No						
0	Mult1	No						
0	Mult2	Yes	DIV	#2xRegs[F4]	Regs[F6]			#5

Entry	Busy	Instruction	State	Destination	Value	Load1	Load2	Load3	Busy	Address
1	No	LD F6, 34(R2)	commit	F6	Mem[load1]	No	No			
2	No	LD F2, 45(R3)	commit	F2	Mem[load2]	No	No			
3	No	MULT F0, F2, F4	commit	F0	#2 x Regs[F4]					
4	Yes	SUBD F8, F6, F2	write	F8	F6 - #2					
5	Yes	DIVD F10, F0, F6	Ex2	F10						
6	Yes	ADDD F6, F8, F2	write	F6	#4 + F2					Reorder Buffer
7										
8										
9										
10										

Reorder #	F0	F2	F4	F6	F8	F10	F12	...	F30
Reorder #				#6	#4	#5			
Busy	No	no	no	Yes	Yes	Yes	no		no

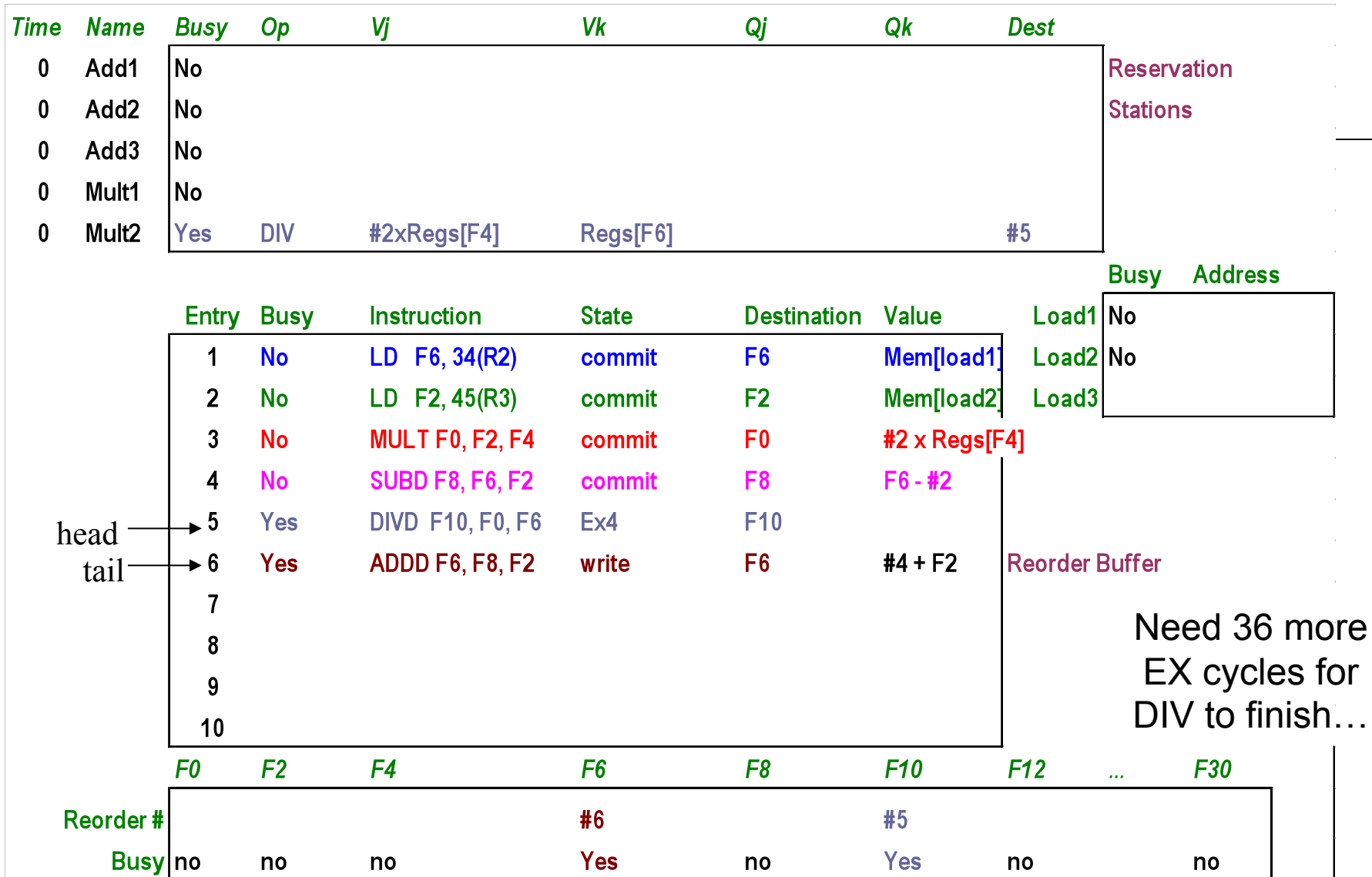
Tomasulo With Reorder Buffer - Cycle 15

Time	Name	Busy	Op	Vj	Vk	Qj	Qk	Dest
0	Add1	No						
0	Add2	No						
0	Add3	No						
0	Mult1	No						
0	Mult2	Yes	DIV	#2xRegs[F4]	Regs[F6]			#5

Entry	Busy	Instruction	State	Destination	Value	Load1	Load2	Load3	Busy	Address
1	No	LD F6, 34(R2)	commit	F6	Mem[load1]	No	No	No		
2	No	LD F2, 45(R3)	commit	F2	Mem[load2]	No	No	No		
3	No	MULT F0, F2, F4	commit	F0	#2 x Regs[F4]					
4	No	SUBD F8, F6, F2	commit	F8	F6 - #2					
5	Yes	DIVD F10, F0, F6	Ex3	F10						
6	Yes	ADDD F6, F8, F2	write	F6	#4 + F2					Reorder Buffer
7										
8										
9										
10										

Reorder #	F0	F2	F4	F6	F8	F10	F12	...	F30
Reorder #				#6		#5			
Busy	no	no	no	Yes	no	Yes	no		no

Tomasulo With Reorder Buffer - Cycle 16



Tomasulo With Reorder Buffer: Summary

Instruction	Issue	Exec Comp	Writeback	Commit
LD F6, 34(R2)	1	2	3	4
LD F2, 45(R3)	2	3	4	5
MULT F0, F2, F4	3	12	13	14
SUBD F8, F6, F2	4	6	7	15
DIVD F10, F0, F6	5	52	53	54
ADDD F6, F8, F2	6	8	9	55

In-order Issue/Commit, Out-of-Order Execution/Writeback

Precise State with ROB

- ROB maintains precise state and allows speculation
 - Waits until precise condition reaches retire/commit stage
 - (Or until branch is noted mis-predicted)
 - Clear ROB, RS, and register status table (Flush)
 - Service exception/Restart from True Branch target
- Need to do similar things with memory ops
 - Called Memory Ordering Buffer (MOB)
 - Completed stores write to MOB then complete (write to memory) in-order (when they reach head of buffer)

Multiple Issue

- Goal: Sustain a CPI of less than 1 by issuing and processing multiple instructions per cycle
- SuperScalar
 - Issue varying number of instructions per clock
 - Statically Scheduled
 - Dynamically Scheduled
- VLIW (EPIC)
 - Issue a fixed number of instructions formatted as one large instruction or instruction “packet”
 - Similar to static-scheduled superscalar

Multiple Issue Choices

Common Name	Issue Structure	Hazard Detection	Scheduling	Examples
Superscalar (static)	Dynamic	Hardware	Static	Sun UltraSPARC II/III
Superscalar (dynamic)	Dynamic	Hardware	Dynamic	IBM POWER2
Superscalar (speculative)	Dynamic	Hardware	Dynamic with speculation	Pentium III/4, MIPS R10K, Alpha 21264, IBM POWER4, HP PA8500
VLIW	Static	Software	Static	Trimedia, i860
EPIC	“mostly” static	mostly software	mostly static	Itanium (IA64)

Multiple Issue Example

	Single Issue Clock Cycle										Multiple Issue Clock Cycle						
	1	2	3	4	5	6	7	8	9	10	1	2	3	4	5	6	7
i	IF	ID	EX	M	WB						IF	ID	EX	M	WB		
$i+1$		IF	ID	EX	M	WB					IF	ID	EX	M	WB		
$i+2$			IF	ID	EX	M	WB					IF	ID	EX	M	WB	
$i+3$				IF	ID	EX	M	WB				IF	ID	EX	M	WB	
$i+4$					IF	ID	EX	M	WB				IF	ID	EX	M	WB
$i+5$						IF	ID	EX	M	WB			IF	ID	EX	M	WB

- Maybe 1 ALU + 1 FP
- 2 ALU + 2 LD/ST + 2 FP
- Many combinations possible – restriction ease implementation

Multiple Issue: Hazards

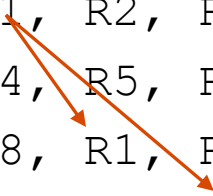
- As usual, we have to deal with the big three hazards:
 - Structural Hazards
 - Data Hazards
 - Control Hazards
- Multiple issue gives:
 - More opportunity for hazards (why?)
 - Larger performance hit from hazards (why?)

Structural Hazards

- If both instructions per cycle are int/float we may need two int ALUs and two FP ALUs
- What about register files?
- This may lead to issue restrictions
 - Compiler/hardware has to manage these restrictions
- 2-issue machines typically do 1 INT/1 FP per cycle
 - Good performance for many apps (+)
 - Hazard Detection is easy (+)
 - No performance boost for non-FP apps (-)

Data Hazards

```
ADD R1, R2, R3
ADD R4, R5, R6
ADD R8, R1, R7
ADD R10, R9, R1
```



- Assume full-bypassing
 - How many stalls for single issue?
 - How many stalls for dual issue?
- Full bypassing?
 - Not easy...

Control Hazards

Branch →

3 Branch Delay Slots

	Multiple Issue Clock Cycle									
	1	2	3	4	5	6	7	8	9	10
i	IF	ID	EX	M	WB					
$i+1$	IF	ID	EX	M	WB					
$i+2$		IF	ID	EX	M	WB				
$i+3$		IF	ID	EX	M	WB				
$i+4$			IF	ID	EX	M	WB			
$i+5$			IF	ID	EX	M	WB			

- Branch stalls bubbles are compounded in n-way machines

Example: Pipeline Problem

IF1	First part of instruction fetch (TLB access)
IF2	Instruction fetch completes (I-cache accessed)
RF	Instruction decoded and register file read
EX	Perform Operation; compute memory address (base+displacement); compute branch target address; compute branch condition
M1	First part of memory access (TLB access)
M2	Memory access completes (D-cache accessed)
WB	Write back results into register file

- How many read/write ports needed?

Pipeline Problem Cont.

	Single Issue Clock Cycle									
	1	2	3	4	5	6	7	8	9	10
i	IF1	IF2	ID	EX	M1	M2	WB			
$i+1$		IF1	IF2	ID	EX	M1	M2	WB		
$i+2$			IF1	IF2	ID	EX	M1	M2	WB	
$i+3$				IF1	IF2	ID	EX	M1	M2	WB
$i+4$					IF1	IF2	ID	EX	M1	M2
$i+5$						IF1	IF2	ID	EX	M1

- What is the branch delay?
- What is the load delay?
- How many adders are needed to prevent structural hazards?
- How many destination RegIDs and comparators are needed for forwarding?

Pipeline Problem Cont.

	Multiple Issue Clock Cycle									
	1	2	3	4	5	6	7	8	9	10
i	IF1	IF2	ID	EX	M1	M2	WB			
$i+1$	IF1	IF2	ID	EX	M1	M2	WB			
$i+2$		IF1	IF2	ID	EX	M1	M2	WB		
$i+3$		IF1	IF2	ID	EX	M1	M2	WB		
$i+4$			IF1	IF2	ID	EX	M1	M2	WB	
$i+5$			IF1	IF2	ID	EX	M1	M2	WB	
$i+6$				IF1	IF2	ID	EX	M1	M2	WB
$i+7$				IF1	IF2	ID	EX	M1	M2	WB
$i+8$					IF1	IF2	ID	EX	M1	M2
$i+9$					IF1	IF2	ID	EX	M1	M2

Branch Delay? Load Delay? Forwarding IDs? Read/Write ports?

Tomasulo + ROB Summary

- Many implementations are very similar
 - Pentium III, PowerPC, etc
- Some limitations
 - Too many value copy operations
 - Register file => RS => ROB => Register File
 - Too many muxes/busses (CDB)
 - Values are coming from everywhere to everywhere else!
 - Reservation Stations mix values(data) and tags(control)
 - Slows down the max clock frequency