



EI 338: Computer Systems Engineering (Operating Systems & Computer Architecture)

Dept. of Computer Science & Engineering

Chentao Wu

wuct@cs.sjtu.edu.cn



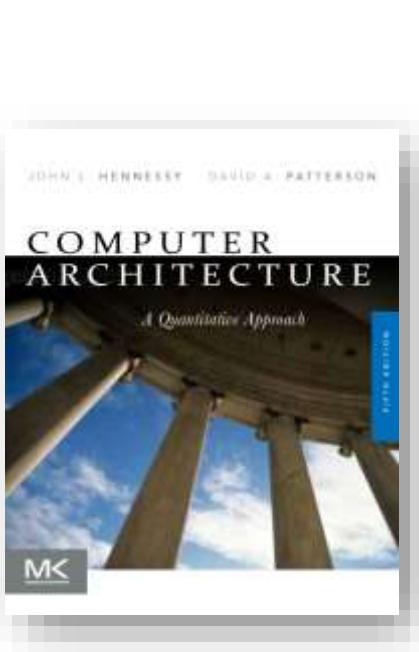
上海交通大学
SHANGHAI JIAO TONG UNIVERSITY

Download lectures

- <ftp://public.sjtu.edu.cn>
- User: wuct
- Password: wuct123456
- <http://www.cs.sjtu.edu.cn/~wuct/cse/>

Computer Architecture

A Quantitative Approach, Fifth Edition



Chapter 3

Instruction-Level Parallelism and Its Exploitation



Outline

- ILP
- Loop unrolling
- Static Branch Prediction
- Dynamic Branch Prediction
- Dynamic Scheduling – Tomasulo's Algorithm
- Reorder Buffer
- CPI less than 1



Recall from Pipelining Review

Pipeline CPI = Ideal pipeline CPI + Structural Stalls + Data Hazard Stalls + Control Stalls

- **Ideal pipeline CPI**: measure of the maximum performance attainable by the implementation
- **Structural hazards**: HW cannot support this combination of instructions
- **Data hazards**: Instruction depends on result of prior instruction still in the pipeline
- **Control hazards**: Caused by delay between the fetching of instructions and decisions about changes in control flow (branches and jumps)

Instruction Level Parallelism

- **Instruction-Level Parallelism (ILP):** overlap the execution of instructions to improve performance
- 2 approaches to exploit ILP:
 - 1) Rely on hardware to help discover and exploit the parallelism **dynamically** (e.g., Pentium 4, AMD Opteron, IBM Power), and
 - 2) Rely on software technology to find parallelism, **statically** at compile-time (e.g., Itanium 2)

Instruction-Level Parallelism (ILP)

- **Basic Block (BB) is quite small**
 - BB: a straight-line code sequence with no branches in except to the entry and no branches out except at the exit
 - average dynamic branch frequency 15% to 25%
=> 4 to 7 instructions execute between a pair of branches
 - Plus instructions in BB likely to depend on each other
- **To obtain substantial performance enhancements, we must exploit ILP across multiple basic blocks**
- **Simplest: loop-level parallelism to exploit parallelism among iterations of a loop. E.g.,**

```
for (i=1; i<=1000; i=i+1)
    x[i] = x[i] + y[i];
```

Loop-Level Parallelism

- Exploit loop-level parallelism by “unrolling loop” either by
 1. dynamic via branch prediction or
 2. static via loop unrolling by compiler
(Another way is vectors, to be covered later)
- Determining instruction dependence is critical to Loop Level Parallelism
- If 2 instructions are
 - parallel, they can execute simultaneously in a pipeline of arbitrary depth without causing any stalls (assuming no structural hazards)
 - dependent, they are not parallel and must be executed in order, although they may often be partially overlapped

Data Dependence and Hazards

- Instr_J is **data dependent** (aka **true dependence**) on Instr_I:
 1. Instr_J tries to read operand before Instr_I writes it
 - ↳ I: add r1, r2, r3
 - ↳ J: sub r4, r1, r3
 2. or Instr_J is data dependent on Instr_K which is dependent on Instr_I
- If two instructions are data dependent, they cannot execute simultaneously or be completely overlapped
- Data dependence in instruction sequence
⇒ data dependence in source code ⇒ effect of original data dependence must be preserved
- If data dependence caused a hazard in pipeline, called a **Read After Write (RAW) hazard**



ILP and Data Dependencies, Hazards

- **HW/SW must preserve program order:** order instructions would execute in if executed sequentially as determined by original source program
 - Dependencies are a property of programs
- Presence of dependence indicates potential for a hazard, but actual hazard and length of any stall is property of the pipeline
- Importance of the data dependencies
 - 1) indicates the possibility of a hazard
 - 2) determines order in which results must be calculated
 - 3) sets an upper bound on how much parallelism can possibly be exploited
- HW/SW goal: exploit parallelism by preserving program order only where it affects the outcome of the program

Name Dependence #1: Anti-dependence

- **Name dependence:** when 2 instructions use same register or memory location, called a **name**, but no flow of data between the instructions associated with that name; **2 versions of name dependence**
- Instr_J writes operand before Instr_I reads it

I: sub r4, **r1**, r3
J: add **r1**, r2, r3
K: mul r6, r1, r7

Called an “anti-dependence” by compiler writers.
This results from reuse of the name “**r1**”

- If anti-dependence caused a hazard in the pipeline, called a **Write After Read (WAR)** hazard

Name Dependence #2: Output dependence

- Instr_J writes operand before Instr_I writes it.

I: sub r1, r4, r3
J: add r1, r2, r3
K: mul r6, r1, r7

- Called an “**output dependence**” by compiler writers
This also results from the reuse of name “r1”
- If anti-dependence caused a hazard in the pipeline,
called a **Write After Write (WAW) hazard**
- Instructions involved in a name dependence can
execute simultaneously if name used in instructions is
changed so instructions do not conflict
 - Register renaming resolves name dependence for regs
 - Either by compiler or by HW

Control Dependencies

- Every instruction is control dependent on some set of branches, and, in general, these control dependencies must be preserved to preserve program order

```
if p1 {  
    S1;  
}  
if p2 {  
    S2;  
}
```

- S1 is control dependent on p1, and S2 is control dependent on p2 but not on p1.

Control Dependence Ignored

- Control dependence need not be preserved
 - willing to execute instructions that should not have been executed, thereby violating the control dependences, if can do so without affecting correctness of the program
- Instead, 2 properties critical to program correctness are
 - 1) exception behavior and
 - 2) data flow

Exception Behavior

- Preserving exception behavior
 - ⇒ any changes in instruction execution order must not change how exceptions are raised in program
(⇒ no new exceptions)
- Example:

DADDU	R2 , R3 , R4
BEQZ	R2 , L1
LW	R1 , 0 (R2)

L1 :

 - (Assume branches not delayed)
- Problem with moving LW before BEQZ?

Data Flow

■ **Data flow:** actual flow of data values among instructions that produce results and those that consume them

- branches make flow dynamic, determine which instruction is supplier of data

■ **Example:**

DADDU R1, R2 , R3

BEQZ R4 , L

DSUBU R1, R5 , R6

L: ...

OR R7 , R1 , R8

- OR depends on DADDU or DSUBU?

Must preserve data flow on execution



Outline

- ILP
- Loop unrolling
- Static Branch Prediction
- Dynamic Branch Prediction
- Dynamic Scheduling – Tomasulo's Algorithm
- Reorder Buffer
- CPI less than 1

Software Techniques - Example

+ This code, add a scalar to a vector:

```
for (i=1000; i>0; i=i-1)  
    x[i] = x[i] + s;
```

- Assume following latencies for all examples
 - Ignore delayed branch in these examples

<i>Instruction producing result</i>	<i>Instruction using result</i>	<i>Latency in cycles</i>	<i>stalls between in cycles</i>
FP ALU op	Another FP ALU op	4	3
FP ALU op	Store double	3	2
Load double	FP ALU op	1	1
Load double	Store double	1	0
Integer op	Integer op	1	0



FP Loop: Where are the Hazards?

- First translate into MIPS code:
 - To simplify, assume 8 is lowest address

```
Loop: L.D      F0,0(R1) ;F0=vector element
      ADD.D    F4,F0,F2 ;add scalar from F2
      S.D      0(R1),F4 ;store result
      DADDUI  R1,R1,-8 ;decrement pointer 8B (DW)
      BNEZ    R1,Loop ;branch R1!=zero
```



FP Loop Showing Stalls

```
1 Loop: L.D      F0 ,0 (R1) ;F0=vector element
2          stall
3 ADD.D  F4 ,F0 ,F2 ;add scalar in F2
4          stall
5          stall
6 S.D    0 (R1) ,F4 ;store result
7 DADDUI R1 ,R1 ,-8 ;decrement pointer 8B (DW)
8          stall       ;assumes can't forward to branch
9 BNEZ   R1 ,Loop   ;branch R1 !=zero
```

<i>Instruction producing result</i>	<i>Instruction using result</i>	<i>Latency in clock cycles</i>
FP ALU op	Another FP ALU op	3
FP ALU op	Store double	2
Load double	FP ALU op	1

- **9 clock cycles: Rewrite code to minimize stalls?**



Revised FP Loop Minimizing Stalls

```
1 Loop: L.D      F0 ,0(R1)
2          DADDUI R1,R1,-8
3          ADD.D   F4 ,F0 ,F2
4          stall
5          stall
6          S.D     8(R1),F4 ;altered offset when move DSUBUI
7          BNEZ    R1,Loop
```

Swap DADDUI and S.D by changing address of S.D

<i>Instruction producing result</i>	<i>Instruction using result</i>	<i>Latency in clock cycles</i>
FP ALU op	Another FP ALU op	3
FP ALU op	Store double	2
Load double	FP ALU op	1

7 clock cycles, but just 3 for execution (L.D, ADD.D,S.D), 4 for loop overhead; How make faster?



Unroll Loop Four Times (straightforward way)

```
1 Loop: L.D      F0,0(R1)           1 cycle stall
3          ADD.D   F4,F0,F2          2 cycles stall
6          S.D     0(R1),F4          ;drop DSUBUI & BNEZ
7          L.D     F6,-8(R1)
9          ADD.D   F8,F6,F2
12         S.D    -8(R1),F8          ;drop DSUBUI & BNEZ
13         L.D    F10,-16(R1)
15         ADD.D   F12,F10,F2
18         S.D    -16(R1),F12          ;drop DSUBUI & BNEZ
19         L.D    F14,-24(R1)
21         ADD.D   F16,F14,F2
24         S.D    -24(R1),F16
25         DADDUI R1,R1,#-32        ;alter to 4*8
26         BNEZ   R1,LOOP
```

Rewrite loop to minimize stalls?

27 clock cycles, or 6.75 per iteration
(Assumes R1 is multiple of 4)

Unrolled Loop Detail

- Do not usually know upper bound of loop
- Suppose it is n , and we would like to unroll the loop to make k copies of the body
- Instead of a single unrolled loop, we generate a pair of consecutive loops:
 - 1st executes $(n \bmod k)$ times and has a body that is the original loop
 - 2nd is the unrolled body surrounded by an outer loop that iterates (n/k) times
- For large values of n , most of the execution time will be spent in the unrolled loop

Unrolled Loop That Minimizes Stalls

```
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     S.D    8(R1),F16 ; 8-32 = -24
14     BNEZ   R1,LOOP
```

14 clock cycles, or 3.5 per iteration



5 Loop Unrolling Decisions

- Requires understanding how one instruction depends on another and how the instructions can be changed or reordered given the dependences:
 1. Determine loop unrolling useful by finding that loop iterations were independent (except for maintenance code)
 2. Use different registers to avoid unnecessary constraints forced by using same registers for different computations
 3. Eliminate the extra test and branch instructions and adjust the loop termination and iteration code
 4. Determine that loads and stores in unrolled loop can be interchanged by observing that loads and stores from different iterations are independent
 - Transformation requires analyzing memory addresses and finding that they do not refer to the same address
 5. Schedule the code, preserving any dependences needed to yield the same result as the original code

3 Limits to Loop Unrolling

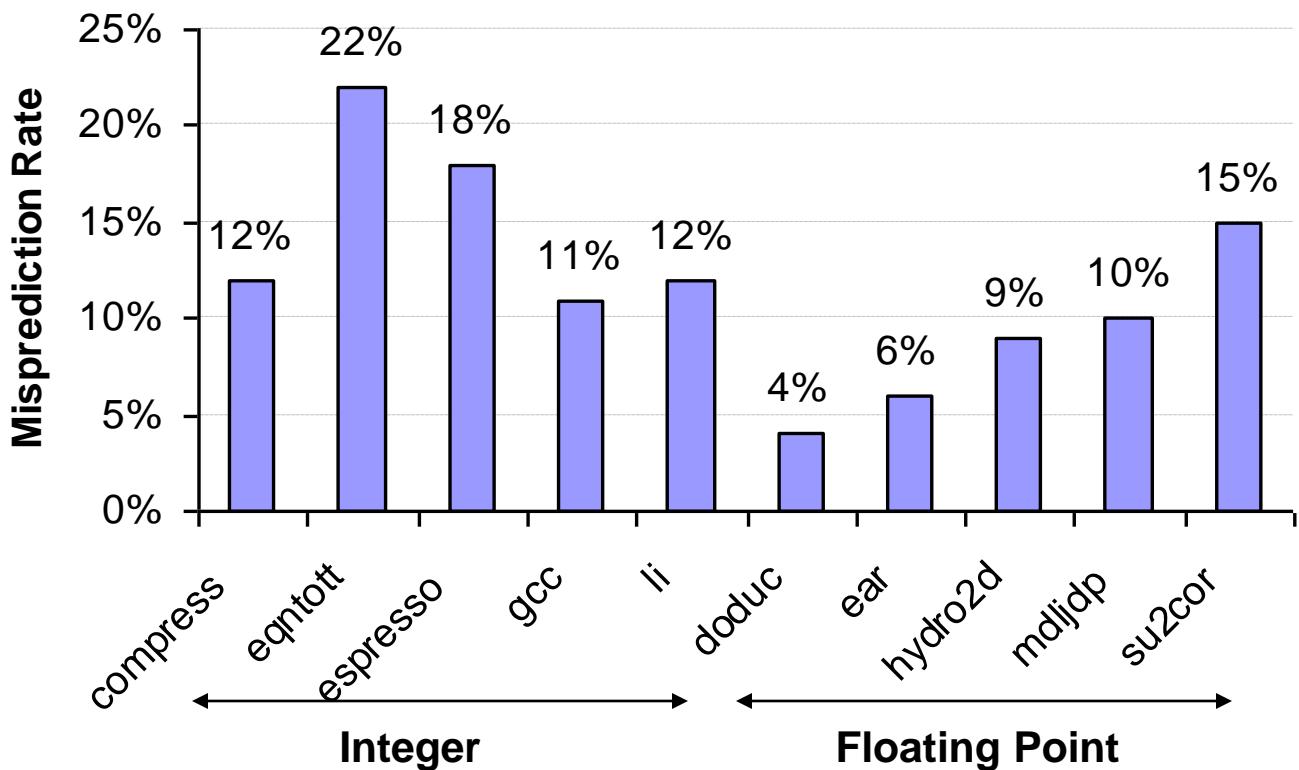
1. Decrease in amount of overhead amortized with each extra unrolling
 - Amdahl's Law
2. Growth in code size
 - For larger loops, concern it increases the instruction cache miss rate
3. Register pressure: potential shortfall in registers created by aggressive unrolling and scheduling
 - If not be possible to allocate all live values to registers, may lose some or all of its advantage
 - Loop unrolling reduces impact of branches on pipeline; another way is **branch prediction**

Outline

- ILP
- Loop unrolling
- **Static Branch Prediction**
- Dynamic Branch Prediction
- Dynamic Scheduling
- Tomasulo Algorithm
- Reorder Buffer
- CPI less than 1

Static Branch Prediction

- To reorder code around branches, need to predict branch statically when compile
- Simplest scheme is to predict a branch as taken
 - Average misprediction = untaken branch frequency = 34% SPEC
- More accurate scheme predicts branches using profile information collected from earlier runs, and modify prediction based on last run:



Outline

- ILP
- Loop unrolling
- Static Branch Prediction
- **Dynamic Branch Prediction**
- Dynamic Scheduling – Tomasulo's Algorithm
- Reorder Buffer
- CPI less than 1

Dynamic Branch Prediction

- 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
- Is dynamic branch prediction better than static branch prediction?
 - Seems to be
 - There are a small number of important branches in programs which have dynamic behavior

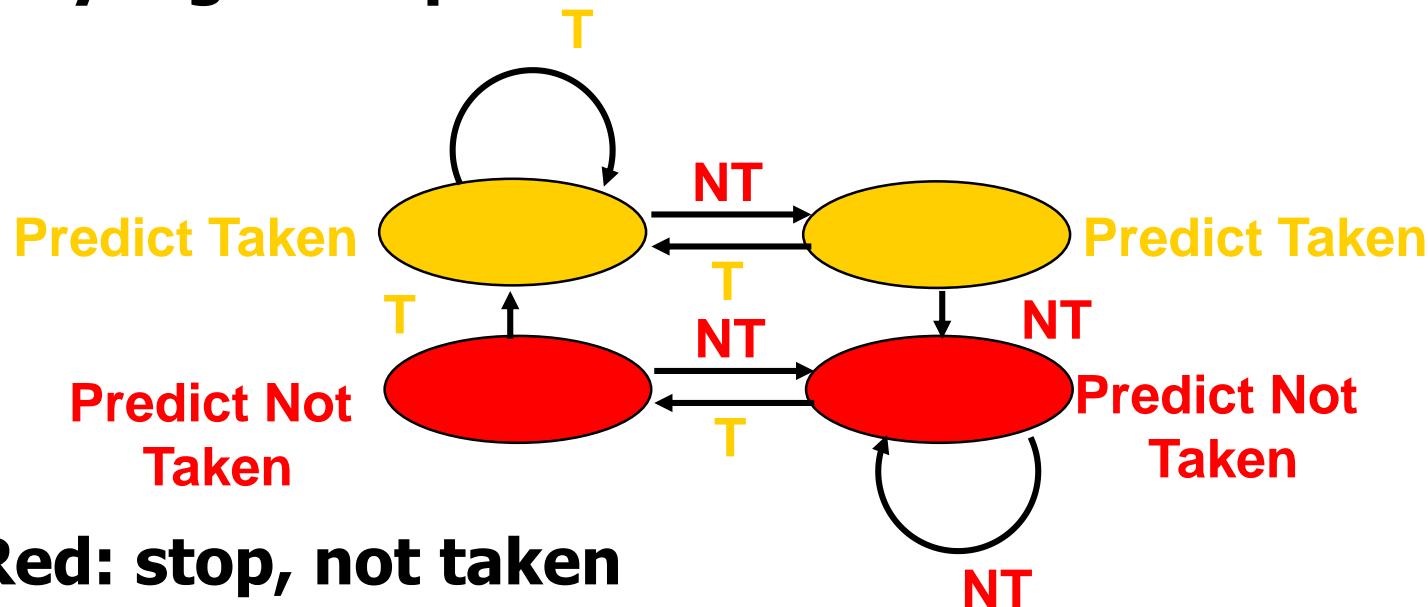


Dynamic Branch Prediction

- **Performance = $f(\text{accuracy}, \text{cost of misprediction})$**
- **Branch History Table:** Lower bits of PC address index table of 1-bit values
 - Says whether or not branch taken last time
 - No address check
- **Problem: in a loop, 1-bit BHT will cause two mispredictions (avg is 9 iterations before exit):**
 - End of loop case, when it exits instead of looping as before
 - First time through loop on **next** time through code, when it predicts exit instead of looping

Dynamic Branch Prediction

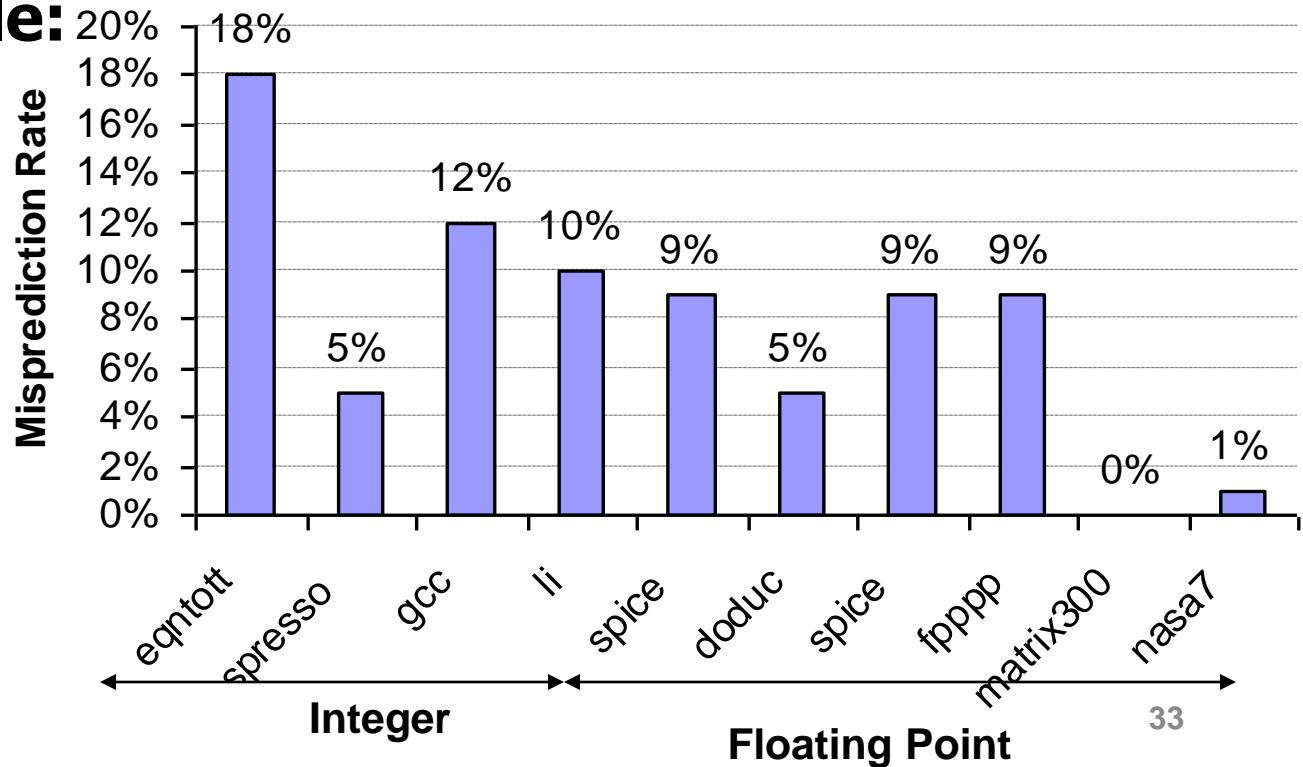
Solution: 2-bit scheme where change prediction only if get misprediction *twice*



- Red: stop, not taken
- Green: go, taken
- Adds *hysteresis* to decision making process

BHT Accuracy

- **Mispredict because either:**
 - Wrong guess for that branch
 - Got branch history of wrong branch when index the table
- **4096 entry table:**



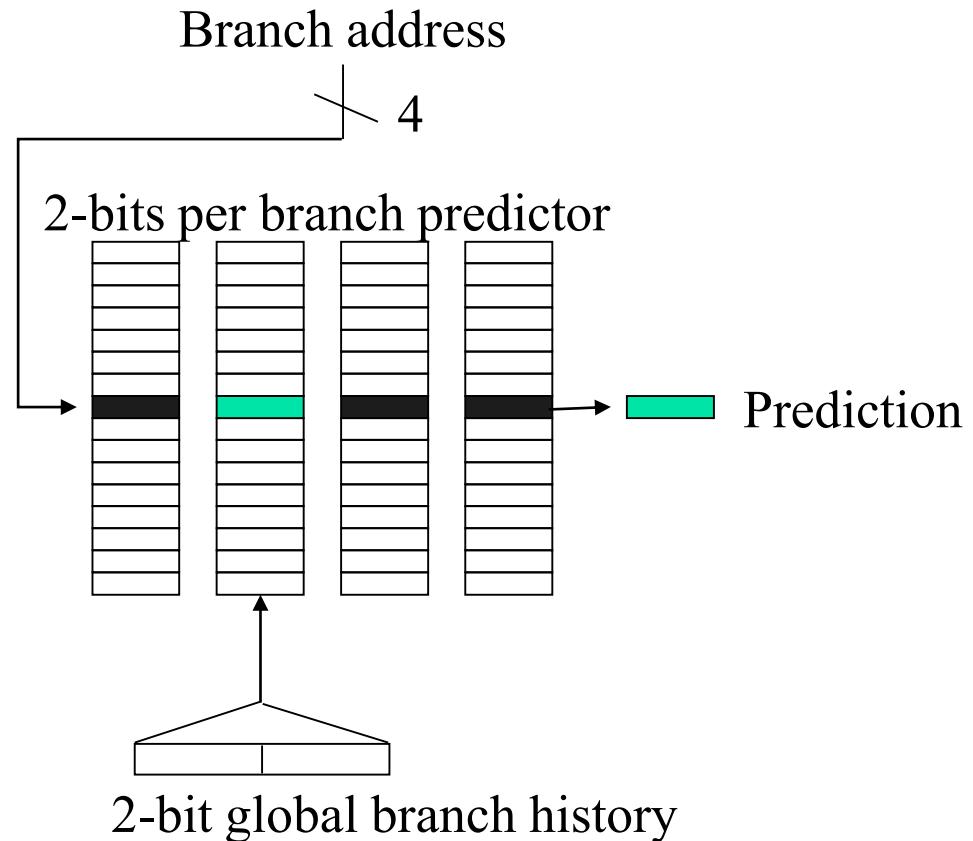
Correlated Branch Prediction

- Idea: record m most recently executed branches as taken or not taken, and use that pattern to select the proper n -bit branch history table
- In general, (m, n) predictor means record last m branches to select between 2^m history tables, each with n -bit counters
 - Thus, old 2-bit BHT is a $(0,2)$ predictor
- Global Branch History: m -bit shift register keeping T/NT status of last m branches.

Correlating Branches

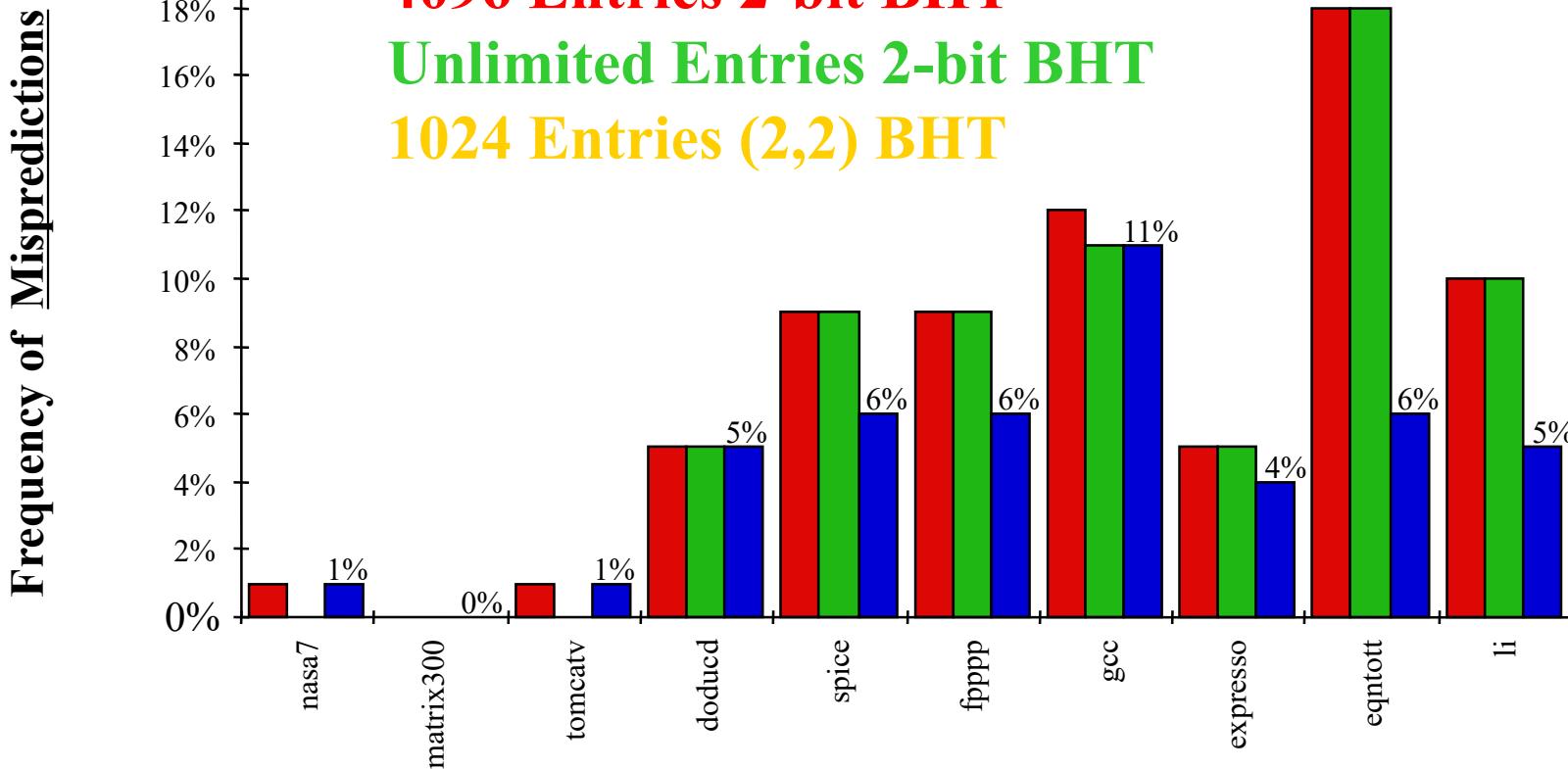
(2,2) predictor

- Behavior of recent branches selects between four predictions of next branch, updating just that prediction





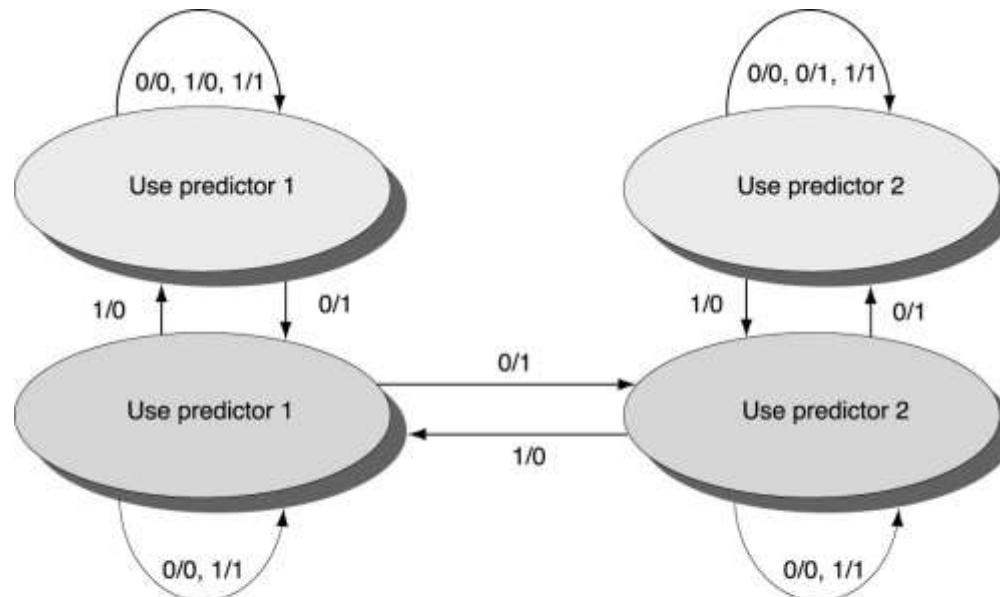
Accuracy of Different Schemes





Tournament Predictors

- Multilevel branch predictor
- Use n -bit saturating counter to choose between predictors
- Usual choice between global and local predictors





Tournament Predictors

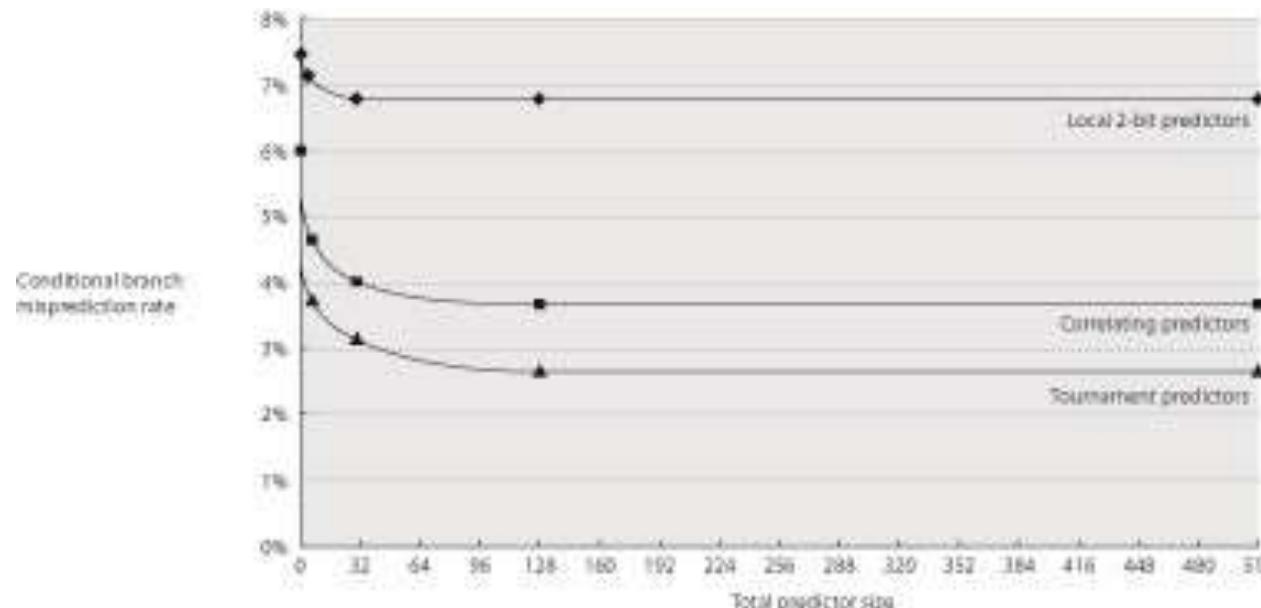
Tournament predictor using, say, 4K 2-bit counters indexed by local branch address. Chooses between:

- **Global predictor**
 - **4K entries index by history of last 12 branches ($2^{12} = 4K$)**
 - **Each entry is a standard 2-bit predictor**
- **Local predictor**
 - **Local history table: 1024 10-bit entries recording last 10 branches, index by branch address**
 - **The pattern of the last 10 occurrences of that particular branch used to index table of 1K entries with 3-bit saturating counters**

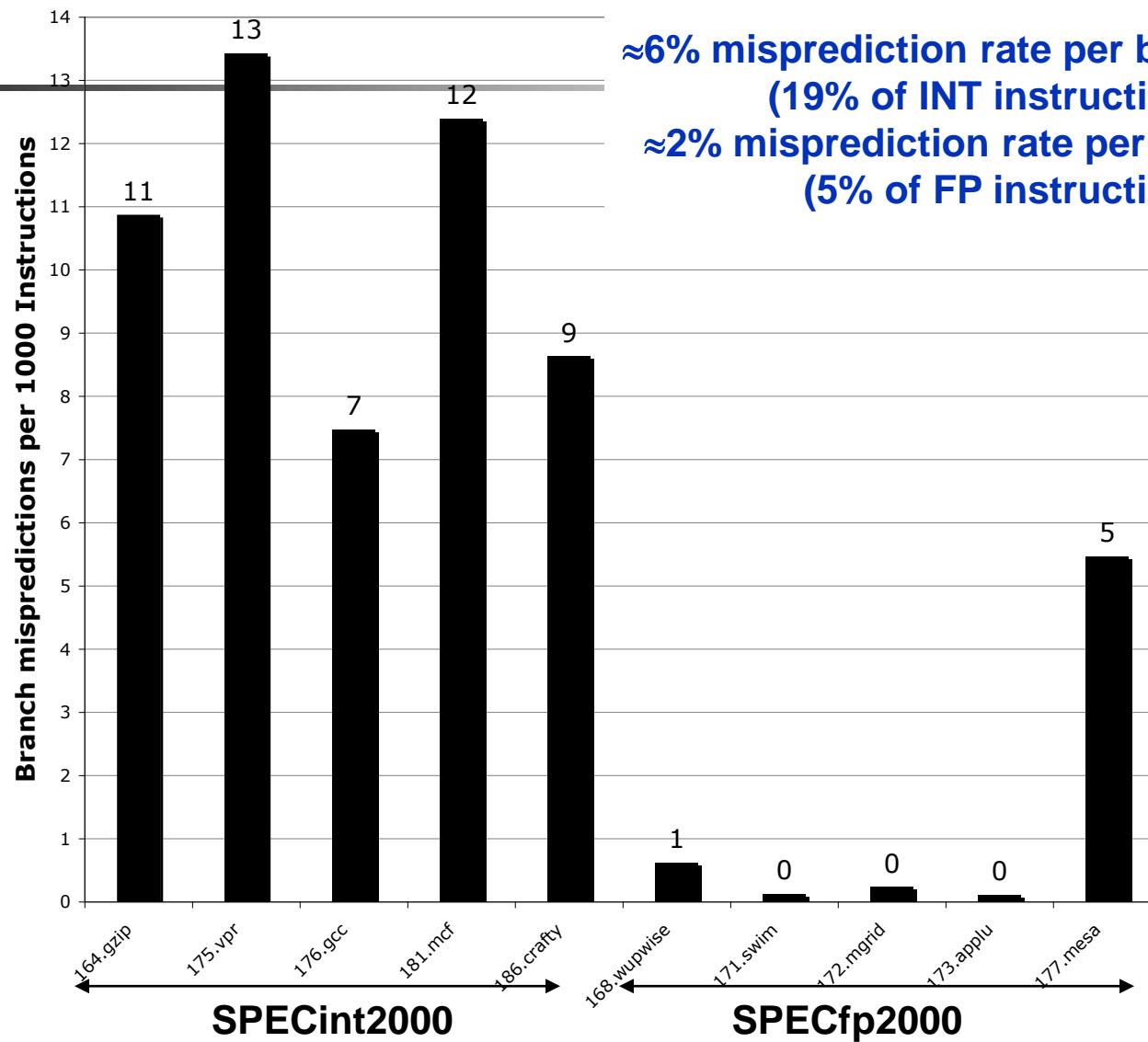


Comparing Predictors

- Advantage of tournament predictor is ability to select the right predictor for a particular branch
 - Particularly crucial for integer benchmarks.
 - A typical tournament predictor will select the global predictor almost 40% of the time for the SPEC integer benchmarks and less than 15% of the time for the SPEC FP benchmarks



Pentium 4 Misprediction Rate (per 1000 instructions, not per branch)

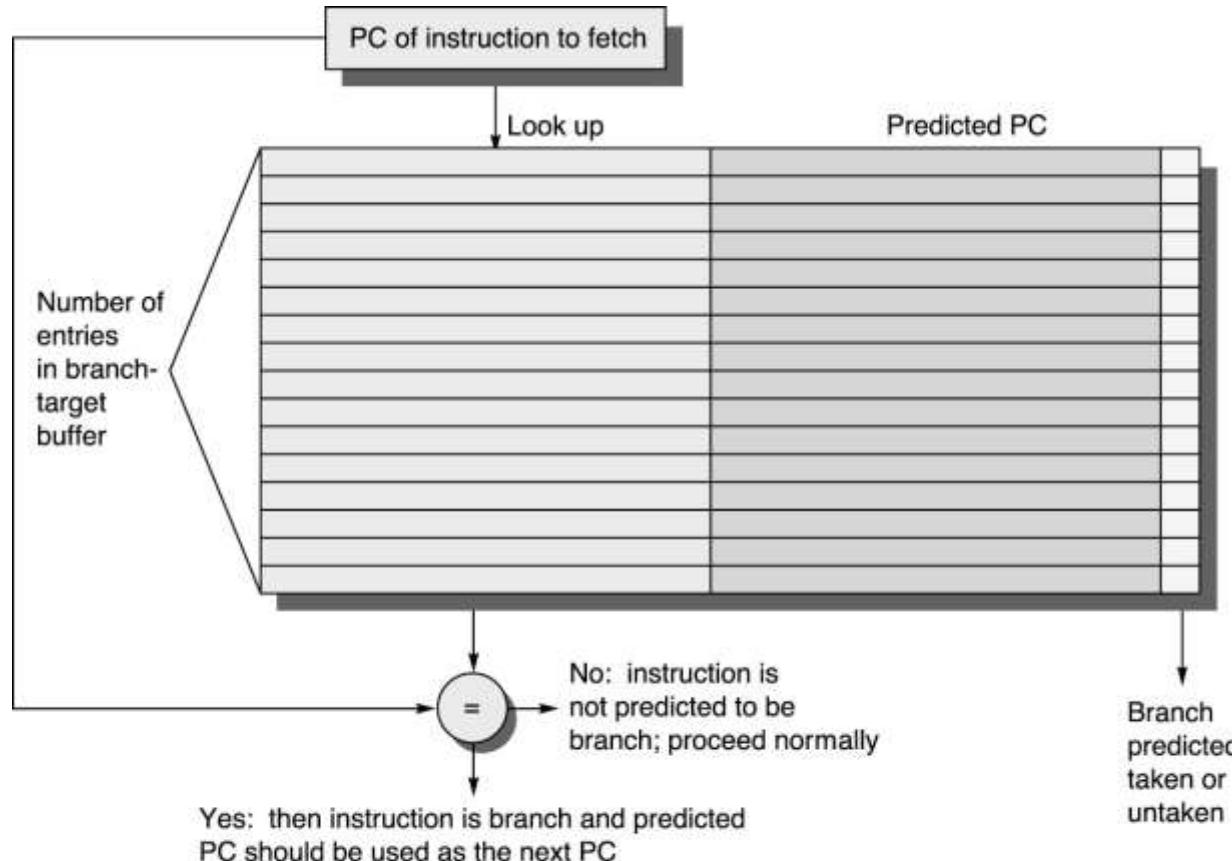


Branch Target Buffers (BTB)

- Branch target calculation is costly and stalls the instruction fetch.
- BTB stores PCs the same way as caches
- The PC of a branch is sent to the BTB
- When a match is found the corresponding Predicted PC is returned
- If the branch was predicted taken, instruction fetch continues at the returned predicted PC



Branch Target Buffers



Dynamic Branch Prediction Summary

- **Prediction becoming important part of execution**
- **Branch History Table:** 2 bits for loop accuracy
- **Correlation:** Recently executed branches correlated with next branch
 - Either different branches (GA)
 - Or different executions of same branches (PA)
- **Tournament predictors take insight to next level, by using multiple predictors**
 - usually one based on global information and one based on local information, and combining them with a selector
 - In 2006, tournament predictors using $\approx 30K$ bits are in processors like the Power5 and Pentium 4
- **Branch Target Buffer: include branch address & prediction**

Outline

- ILP
- Loop unrolling
- Static Branch Prediction
- Dynamic Branch Prediction
- **Dynamic Scheduling – Tomasulo's Algorithm**
- Reorder Buffer
- CPI less than 1

Advantages of Dynamic Scheduling

- **Dynamic scheduling** - hardware rearranges the instruction execution to reduce stalls while maintaining data flow and exception behavior
- It handles cases when dependences unknown at compile time
 - it allows the processor to tolerate unpredictable delays such as cache misses, by executing other code while waiting for the miss to resolve
- It allows code that compiled for one pipeline to run efficiently on a different pipeline
- It simplifies the compiler
- **Hardware speculation**, a technique with significant performance advantages, builds on dynamic scheduling

HW Schemes: Instruction Parallelism

- Key idea: Allow instructions behind stall to proceed
 - DIVD F0, F2, F4
 - ADDD F10, F0, F8
 - SUBD F12, F8, F14
- Enables **out-of-order execution** and allows **out-of-order completion** (e.g., SUBD)
 - In a dynamically scheduled pipeline, all instructions still pass through issue stage in order (**in-order issue**)
- Will distinguish when an instruction ***begins execution*** and when it ***completes execution***; between 2 times, the instruction is ***in execution***
- Note: Dynamic execution creates WAR and WAW hazards and makes exceptions harder



Dynamic Scheduling Step 1

- Simple pipeline had 1 stage to check both structural and data hazards: Instruction Decode (ID), also called Instruction Issue
- Split the ID pipe stage of simple 5-stage pipeline into 2 stages:
- ***Issue***—Decode instructions, check for structural hazards
- ***Read operands***—Wait until no data hazards, then read operands

A Dynamic Algorithm: Tomasulo's

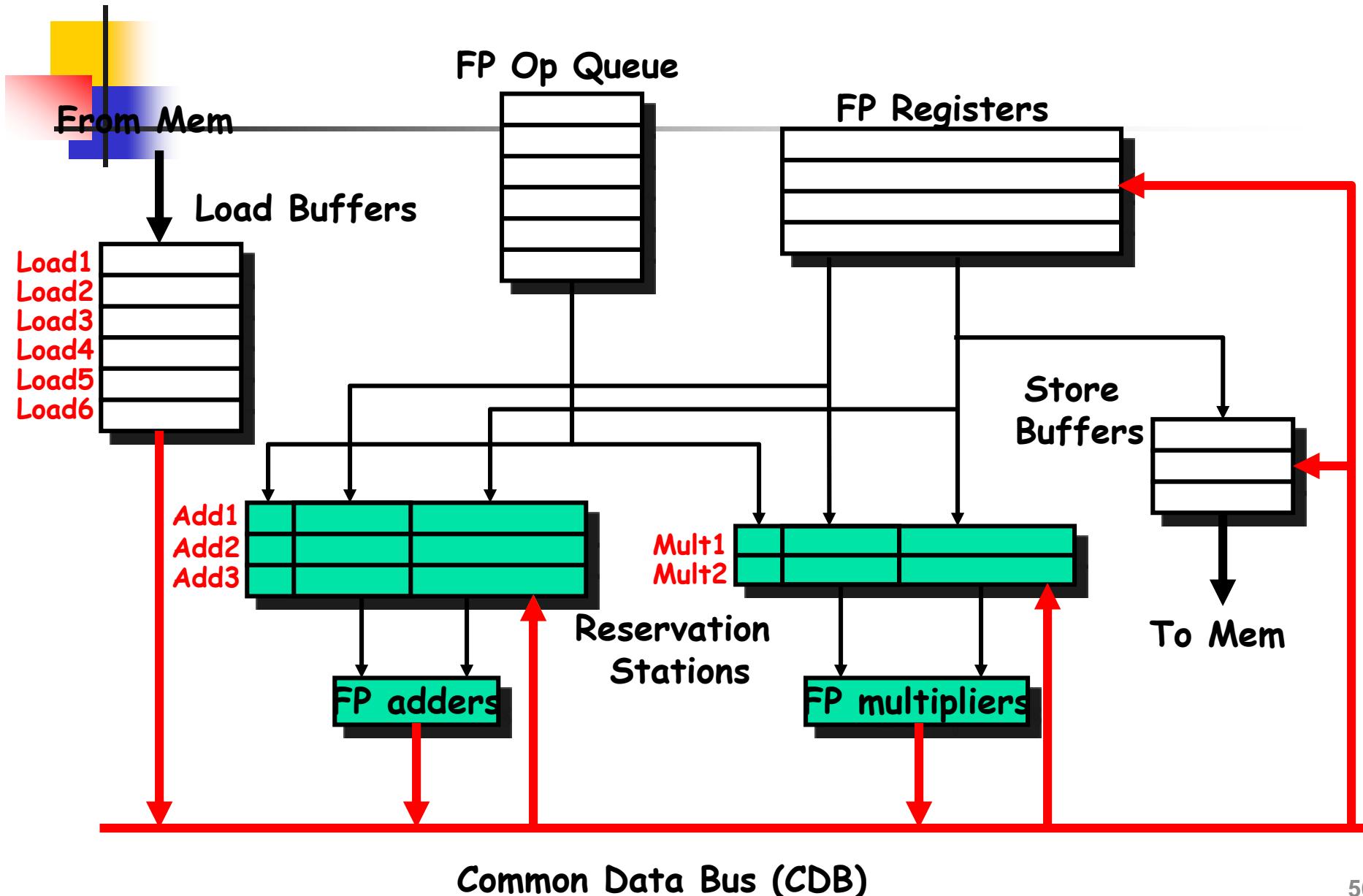
- **For IBM 360/91 (before caches!)**
 - ⇒ Long memory latency
- **Goal: High Performance without special compilers**
- **Small number of floating point registers (4 in 360) prevented interesting compiler scheduling of operations**
 - This led Tomasulo to try to figure out how to get more effective registers — **renaming in hardware!**
- **Why Study 1966 Computer?**
- **The descendants of this have flourished!**
 - Alpha 21264, Pentium 4, AMD Opteron, Power 5, ...

Tomasulo Algorithm

Control & buffers distributed with Function Units (FU)

- FU buffers called "reservation stations"; have pending operands
- Registers in instructions replaced by values or pointers to reservation stations(RS); called register renaming ;
 - Renaming avoids WAR, WAW hazards
 - More reservation stations than registers, so can do optimizations compilers can't
- Results to FU from RS, not through registers, over Common Data Bus that broadcasts results to all FUs
 - Avoids RAW hazards by executing an instruction only when its operands are available
- Load and Stores treated as FUs with RSs as well
- Integer instructions can go past branches (predict taken), allowing FP ops beyond basic block in FP queue

Tomasulo Organization





Reservation Station Components

Op: Operation to perform in the unit (e.g., + or -)

V_j, V_k: Value of Source operands

- Store buffers has V field, result to be stored

Q_j, Q_k: Reservation stations producing source registers (value to be written)

- Note: Q_j, Q_k=0 => ready
- Store buffers only have Q_i for RS producing result

Busy: Indicates reservation station or FU is busy

Register result status—Indicates which functional unit will write each register, if one exists. Blank when no pending instructions that will write that register.



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. Execute—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
- **Example speed:**
3 clocks for Fl .pt. +,-; 10 for * ; 40 clks for /

Tomasulo Example

Instruction stream

Instruction status:

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

Busy	Address
No	
No	
No	

3 Load/Buffers

Reservation Stations:

Time	Name	Busy	Op	S1	S2	RS	RS
	Add1	No					
	Add2	No					
	Add3	No					
	Mult1	No					
	Mult2	No					

3 FP Adder R.S.
2 FP Mult R.S.

Register result status:

Clock

0

Clock cycle
counter

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

Tomasulo Example Cycle 1

Instruction status:

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

	Busy	Address
Load1	Yes	34+R2
Load2	No	
Load3	No	

Reservation Stations:

Time	Name	Busy	Op	S1	S2	RS	RS
				Vj	Vk	Qj	Qk
	Add1	No					
	Add2	No					
	Add3	No					
	Mult1	No					
	Mult2	No					

Register result status:

Clock	F0	F2	F4	F6	F8	F10	F12	...	F30
1				Load1					

Tomasulo Example Cycle 2

Instruction status:

Instruction	j	k	Issue	Exec	Write
				Comp	Result
LD	F6	34+	R2	1	
LD	F2	45+	R3	2	
MULTD	F0	F2	F4		
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	S2	RS	RS
				Vj	Vk	Qj	Qk
	Add1	No					
	Add2	No					
	Add3	No					
	Mult1	No					
	Mult2	No					

Register result status:

Clock	F0	F2	F4	F6	F8	F10	F12	...	F30
2	FU	Load2		Load1					

Note: Can have multiple loads outstanding

Tomasulo Example Cycle 3

Instruction status:

Instruction	j	k	Issue	Exec	Write	Busy	Address
				Comp	Result		
LD	F6	34+	R2	1	3	Load1	Yes 34+R2
LD	F2	45+	R3	2		Load2	Yes 45+R3
MULTD	F0	F2	F4	3		Load3	No
SUBD	F8	F6	F2				
DIVD	F10	F0	F6				
ADDD	F6	F8	F2				

Reservation Stations:

Time	Name	Busy	Op	S1	S2	RS	RS
				Vj	Vk	Qj	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; MULT issued
- Load1 completing; what is waiting for Load1?

Tomasulo Example Cycle 4

Instruction status:

Instruction	j	k	Exec			Write	Busy	Address
			Issue	Comp	Result			
LD	F6	34+	R2	1	3	4	Load1	No
LD	F2	45+	R3	2	4		Load2	Yes 45+R3
MULTD	F0	F2	F4	3			Load3	No
SUBD	F8	F6	F2		4			
DIVD	F10	F0	F6					
ADDD	F6	F8	F2					

Reservation Stations:

Time	Name	Busy	Op	S1		RS	
				Vi	Vk	Oi	Ok
	Add1	Yes	SUBD	M(A1)			Load2
	Add2	No					
	Add3	No					
	Mult1	Yes	MULTD		R(F4)	Load2	
	Mult2	No					

Register result status:

Clock	F0	F2	F4	F6	F8	F10	F12	...	F30
4	FU	Mult1	Load2		M(A1)	Add1			

- Load2 completing; what is waiting for Load2?

Tomasulo Example Cycle 5

Instruction status:

Instruction	<i>j</i>	<i>k</i>	<i>Issue</i>	<i>Exec</i>	<i>Write</i>	<i>Busy</i>	<i>Address</i>
				<i>Comp</i>	<i>Result</i>		
LD	F6	34+	R2	1	3	4	Load1 No
LD	F2	45+	R3	2	4	5	Load2 No
MULTD	F0	F2	F4	3			Load3 No
SUBD	F8	F6	F2	4			
DIVD	F10	F0	F6	5			
ADDD	F6	F8	F2				

Reservation Stations:

<i>Time</i>	<i>Name</i>	<i>Busy</i>	<i>Op</i>	<i>Vj</i>	<i>Vk</i>	<i>Qj</i>	<i>Qk</i>
2	Add1	Yes	SUBD	M(A1)	M(A2)		
	Add2	No					
	Add3	No					
10	Mult1	Yes	MULTD	M(A2)	R(F4)		
	Mult2	Yes	DIVD		M(A1)	Mult1	

Register result status:

Clock	<i>F0</i>	<i>F2</i>	<i>F4</i>	<i>F6</i>	<i>F8</i>	<i>F10</i>	<i>F12</i>	...	<i>F30</i>
5	<i>FU</i>	Mult1 M(A2)		M(A1)	Add1	Mult2			

- Timer starts down for Add1, Mult1

Tomasulo Example Cycle 6

Instruction status:

Instruction	<i>j</i>	<i>k</i>	<i>Issue</i>	<i>Exec</i>	<i>Write</i>	<i>Busy</i>	<i>Address</i>
				<i>Comp</i>	<i>Result</i>		
LD	F6	34+	R2	1	3	4	Load1 No
LD	F2	45+	R3	2	4	5	Load2 No
MULTD	F0	F2	F4	3			Load3 No
SUBD	F8	F6	F2	4			
DIVD	F10	F0	F6	5			
ADDD	F6	F8	F2	6			

Reservation Stations:

Time	Name	<i>Busy</i>	<i>Op</i>	<i>S1</i>	<i>S2</i>	<i>RS</i>	<i>RS</i>
				<i>Vj</i>	<i>Vk</i>	<i>Qj</i>	<i>Qk</i>
1	Add1	Yes	SUBD	M(A1)	M(A2)		
	Add2	Yes	ADDD		M(A2)	Add1	
	Add3	No					
9	Mult1	Yes	MULTD	M(A2)	R(F4)		
	Mult2	Yes	DIVD		M(A1)	Mult1	

Register result status:

Clock	<i>F0</i>	<i>F2</i>	<i>F4</i>	<i>F6</i>	<i>F8</i>	<i>F10</i>	<i>F12</i>	...	<i>F30</i>
6	<i>FU</i>	Mult1	M(A2)		Add2	Add1	Mult2		

- Issue ADDD here despite name dependency on F6?

Tomasulo Example Cycle 7

Instruction status:

Instruction	<i>j</i>	<i>k</i>	<i>Issue</i>	<i>Exec</i>	<i>Write</i>	<i>Busy</i>	<i>Address</i>
				<i>Comp</i>	<i>Result</i>		
LD	F6	34+	R2	1	3	4	Load1 No
LD	F2	45+	R3	2	4	5	Load2 No
MULTD	F0	F2	F4	3			Load3 No
SUBD	F8	F6	F2	4	7		
DIVD	F10	F0	F6	5			
ADDD	F6	F8	F2	6			

Reservation Stations:

Time	Name	Busy	<i>Op</i>	<i>S1</i>	<i>S2</i>	<i>RS</i>	<i>RS</i>
				<i>Vj</i>	<i>Vk</i>	<i>Qj</i>	<i>Qk</i>
0	Add1	Yes	SUBD	M(A1)	M(A2)		
	Add2	Yes	ADDD		M(A2)	Add1	
	Add3	No					
8	Mult1	Yes	MULTD	M(A2)	R(F4)		
	Mult2	Yes	DIVD		M(A1)	Mult1	

Register result status:

Clock	<i>F0</i>	<i>F2</i>	<i>F4</i>	<i>F6</i>	<i>F8</i>	<i>F10</i>	<i>F12</i>	...	<i>F30</i>
7	<i>FU</i>	Mult1 M(A2)		Add2	Add1	Mult2			

- Add1 (SUBD) completing; what is waiting for it?

Tomasulo Example Cycle 8

Instruction status:

Instruction	<i>j</i>	<i>k</i>	Exec Write			Busy	Address	
			Issue	Comp	Result			
LD	F6	34+	R2	1	3	4	Load1	No
LD	F2	45+	R3	2	4	5	Load2	No
MULTD	F0	F2	F4	3			Load3	No
SUBD	F8	F6	F2	4	7	8		
DIVD	F10	F0	F6	5				
ADDD	F6	F8	F2	6				

Reservation Stations:

Time	Name	Busy	Op	<i>S₁</i>	<i>S₂</i>	<i>RS</i>	<i>RS</i>
	Add1	No					
2	Add2	Yes	ADDD	(M-M)	M(A2)		
	Add3	No					
7	Mult1	Yes	MULTD	M(A2)	R(F4)		
	Mult2	Yes	DIVD		M(A1)	Mult1	

Register result status:

Clock	<i>F₀</i>	<i>F₂</i>	<i>F₄</i>	<i>F₆</i>	<i>F₈</i>	<i>F₁₀</i>	<i>F₁₂</i>	...	<i>F₃₀</i>
8		Mult1	M(A2)		Add2	(M-M)	Mult2		

Tomasulo Example Cycle 9

Instruction status:

Instruction	<i>j</i>	<i>k</i>	<i>Issue</i>	<i>Exec</i>	<i>Write</i>	<i>Busy</i>	<i>Address</i>
				<i>Comp</i>	<i>Result</i>		
LD	F6	34+	R2	1	3	4	Load1 No
LD	F2	45+	R3	2	4	5	Load2 No
MULTD	F0	F2	F4	3			Load3 No
SUBD	F8	F6	F2	4	7	8	
DIVD	F10	F0	F6	5			
ADDD	F6	F8	F2	6			

Reservation Stations:

<i>Time</i>	<i>Name</i>	<i>Busy</i>	<i>Op</i>	<i>S1</i>	<i>S2</i>	<i>RS</i>	<i>RS</i>
				<i>Vj</i>	<i>Vk</i>	<i>Qj</i>	<i>Qk</i>
	Add1	No					
1	Add2	Yes	ADDD	(M-M)	M(A2)		
	Add3	No					
6	Mult1	Yes	MULTD	M(A2)	R(F4)		
	Mult2	Yes	DIVD		M(A1)	Mult1	

Register result status:

Clock		<i>F0</i>	<i>F2</i>	<i>F4</i>	<i>F6</i>	<i>F8</i>	<i>F10</i>	<i>F12</i>	...	<i>F30</i>
9	<i>FU</i>	Mult1	M(A2)		Add2	(M-M)	Mult2			

Tomasulo Example Cycle 10

Instruction status:

Instruction	<i>j</i>	<i>k</i>	<i>Issue</i>	<i>Exec</i>	<i>Write</i>	<i>Busy</i>	<i>Address</i>
				<i>Comp</i>	<i>Result</i>		
LD	F6	34+	R2	1	3	4	Load1 No
LD	F2	45+	R3	2	4	5	Load2 No
MULTD	F0	F2	F4	3			Load3 No
SUBD	F8	F6	F2	4	7	8	
DIVD	F10	F0	F6	5			
ADDD	F6	F8	F2	6	10		

Reservation Stations:

Time	Name	Busy	<i>S1</i>	<i>S2</i>	<i>RS</i>	<i>RS</i>	
			<i>Op</i>	<i>Vj</i>	<i>Vk</i>	<i>Qj</i>	<i>Qk</i>
	Add1	No					
0	Add2	Yes	ADDD	(M-M)	M(A2)		
	Add3	No					
5	Mult1	Yes	MULTD	M(A2)	R(F4)		
	Mult2	Yes	DIVD		M(A1)	Mult1	

Register result status:

Clock	<i>F0</i>	<i>F2</i>	<i>F4</i>	<i>F6</i>	<i>F8</i>	<i>F10</i>	<i>F12</i>	...	<i>F30</i>
10	<i>FU</i>	Mult1	M(A2)		Add2	(M-M)	Mult2		

- Add2 (ADDD) completing; what is waiting for it?

Tomasulo Example Cycle 11

Instruction status:

Instruction	j	k	Issue	Exec		Result	Busy	Address
				Comp	Write			
LD	F6	34+	R2	1	3	4	Load1	No
LD	F2	45+	R3	2	4	5	Load2	No
MULTD	F0	F2	F4	3			Load3	No
SUBD	F8	F6	F2	4	7	8		
DIVD	F10	F0	F6	5				
ADDD	F6	F8	F2	6	10	11		

Reservation Stations:

Time	Name	Busy	S1		S2		RS	
			Op	Vj	Vk	Qj	Qk	
	Add1	No						
	Add2	No						
	Add3	No						
4	Mult1	Yes	MULTD	M(A2)	R(F4)			
	Mult2	Yes	DIVD		M(A1)	Mult1		

Register result status:

Clock		F0	F2	F4	F6	F8	F10	F12	...	F30
11	FU	Mult1	M(A2)		(M-M+M)	(M-M)	Mult2			

- Write result of ADDD here?
- All quick instructions complete in this cycle!

Tomasulo Example Cycle 12

Instruction status:

Instruction	<i>j</i>	<i>k</i>	<i>Issue</i>	<i>Exec</i>	<i>Write</i>	<i>Busy</i>	<i>Address</i>
				<i>Comp</i>	<i>Result</i>		
LD	F6	34+	R2	1	3	4	Load1 No
LD	F2	45+	R3	2	4	5	Load2 No
MULTD	F0	F2	F4	3			Load3 No
SUBD	F8	F6	F2	4	7	8	
DIVD	F10	F0	F6	5			
ADDD	F6	F8	F2	6	10	11	

Reservation Stations:

<i>Time</i>	<i>Name</i>	<i>Busy</i>	<i>Op</i>	<i>S1</i>	<i>S2</i>	<i>RS</i>	<i>RS</i>
				<i>Vj</i>	<i>Vk</i>	<i>Qj</i>	<i>Qk</i>
	Add1	No					
	Add2	No					
	Add3	No					
3	Mult1	Yes	MULTD M(A2) R(F4)				
	Mult2	Yes	DIVD		M(A1) Mult1		

Register result status:

Clock	<i>F0</i>	<i>F2</i>	<i>F4</i>	<i>F6</i>	<i>F8</i>	<i>F10</i>	<i>F12</i>	...	<i>F30</i>
<i>FU</i>	Mult1	M(A2)		(M-M+M)	(M-M)	Mult2			
12									

Tomasulo Example Cycle 13

Instruction status:

Instruction	<i>j</i>	<i>k</i>	<i>Issue</i>	<i>Exec</i>	<i>Write</i>	<i>Busy</i>	<i>Address</i>
				<i>Comp</i>	<i>Result</i>		
LD	F6	34+	R2	1	3	4	Load1 No
LD	F2	45+	R3	2	4	5	Load2 No
MULTD	F0	F2	F4	3			Load3 No
SUBD	F8	F6	F2	4	7	8	
DIVD	F10	F0	F6	5			
ADDD	F6	F8	F2	6	10	11	

Reservation Stations:

<i>Time</i>	<i>Name</i>	<i>Busy</i>	<i>Op</i>	<i>S1</i>	<i>S2</i>	<i>RS</i>	<i>RS</i>
				<i>Vj</i>	<i>Vk</i>	<i>Qj</i>	<i>Qk</i>
	Add1	No					
	Add2	No					
	Add3	No					
2	Mult1	Yes	MULTD M(A2) R(F4)				
	Mult2	Yes	DIVD		M(A1) Mult1		

Register result status:

Clock	<i>F0</i>	<i>F2</i>	<i>F4</i>	<i>F6</i>	<i>F8</i>	<i>F10</i>	<i>F12</i>	...	<i>F30</i>
<i>13</i>	<i>FU</i>	Mult1 M(A2)		(M-M+M)(M-M)	Mult2				

Tomasulo Example Cycle 14

Instruction status:

Instruction	<i>j</i>	<i>k</i>	<i>Issue</i>	<i>Exec</i>	<i>Write</i>	<i>Busy</i>	<i>Address</i>
				<i>Comp</i>	<i>Result</i>		
LD	F6	34+	R2	1	3	4	Load1 No
LD	F2	45+	R3	2	4	5	Load2 No
MULTD	F0	F2	F4	3			Load3 No
SUBD	F8	F6	F2	4	7	8	
DIVD	F10	F0	F6	5			
ADDD	F6	F8	F2	6	10	11	

Reservation Stations:

<i>Time</i>	<i>Name</i>	<i>Busy</i>	<i>Op</i>	<i>S1</i>	<i>S2</i>	<i>RS</i>	<i>RS</i>
				<i>Vj</i>	<i>Vk</i>	<i>Qj</i>	<i>Qk</i>
	Add1	No					
	Add2	No					
	Add3	No					
1	Mult1	Yes	MULTD	M(A2)	R(F4)		
	Mult2	Yes	DIVD		M(A1)	Mult1	

Register result status:

Clock	<i>F0</i>	<i>F2</i>	<i>F4</i>	<i>F6</i>	<i>F8</i>	<i>F10</i>	<i>F12</i>	...	<i>F30</i>
14	<i>FU</i>	Mult1	M(A2)	(M-M+M)	(M-M)	Mult2			

Tomasulo Example Cycle 15

Instruction status:

Instruction	j	k	Issue	Exec	Write	Busy	Address	
				Comp	Result			
LD	F6	34+	R2	1	3	4	Load1	No
LD	F2	45+	R3	2	4	5	Load2	No
MULTD	F0	F2	F4	3	15		Load3	No
SUBD	F8	F6	F2	4	7	8		
DIVD	F10	F0	F6	5				
ADDD	F6	F8	F2	6	10	11		

Reservation Stations:

Time	Name	Busy	Op	S1	S2	RS	RS
				Vj	Vk	Qj	Qk
	Add1	No					
	Add2	No					
	Add3	No					
0	Mult1	Yes	MULTD M(A2)	R(F4)			
	Mult2	Yes	DIVD		M(A1)	Mult1	

Register result status:

Clock	F0	F2	F4	F6	F8	F10	F12	...	F30
15	FU	Mult1	M(A2)	(M-M+N(M-M))	Mult2				

- Mult1 (MULTD) completing; what is waiting for it?

Tomasulo Example Cycle 16

Instruction status:

Instruction	<i>j</i>	<i>k</i>	<i>Issue</i>	<i>Exec</i>	<i>Write</i>	<i>Busy</i>	<i>Address</i>
				<i>Comp</i>	<i>Result</i>		
LD	F6	34+	R2	1	3	4	Load1
LD	F2	45+	R3	2	4	5	Load2
MULTD	F0	F2	F4	3	15	16	Load3
SUBD	F8	F6	F2	4	7	8	
DIVD	F10	F0	F6	5			
ADDD	F6	F8	F2	6	10	11	

Reservation Stations:

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

Register result status:

Clock	<i>F0</i>	<i>F2</i>	<i>F4</i>	<i>F6</i>	<i>F8</i>	<i>F10</i>	<i>F12</i>	...	<i>F30</i>
16	<i>FU</i>	M*F4	M(A2)	(M-M+M(M-M))	Mult2				

- Just waiting for Mult2 (DIVD) to complete



**Faster than light
computation
(skip a couple of cycles)**

Tomasulo Example Cycle 55

Instruction status:

Instruction	<i>j</i>	<i>k</i>	<i>Issue</i>	<i>Exec</i>	<i>Write</i>	<i>Busy</i>	<i>Address</i>
				<i>Comp</i>	<i>Result</i>		
LD	F6	34+	R2	1	3	4	Load1 No
LD	F2	45+	R3	2	4	5	Load2 No
MULTD	F0	F2	F4	3	15	16	Load3 No
SUBD	F8	F6	F2	4	7	8	
DIVD	F10	F0	F6	5			
ADDD	F6	F8	F2	6	10	11	

Reservation Stations:

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

Register result status:

Clock	<i>F0</i>	<i>F2</i>	<i>F4</i>	<i>F6</i>	<i>F8</i>	<i>F10</i>	<i>F12</i>	...	<i>F30</i>
<i>FU</i>	M*F4	M(A2)		(M-M+M	(M-M)	Mult2			
55									

Tomasulo Example Cycle 56

Instruction status:

Instruction	<i>j</i>	<i>k</i>	<i>Issue</i>	<i>Exec</i>	<i>Write</i>	<i>Busy</i>	<i>Address</i>
				<i>Comp</i>	<i>Result</i>		
LD	F6	34+	R2	1	3	4	Load1
LD	F2	45+	R3	2	4	5	Load2
MULTD	F0	F2	F4	3	15	16	Load3
SUBD	F8	F6	F2	4	7	8	
DIVD	F10	F0	F6	5	56		
ADDD	F6	F8	F2	6	10	11	

Reservation Stations:

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

Register result status:

Clock	<i>F0</i>	<i>F2</i>	<i>F4</i>	<i>F6</i>	<i>F8</i>	<i>F10</i>	<i>F12</i>	...	<i>F30</i>
56	<i>FU</i>	M*F4	M(A2)	(M-M+M)	(M-M)	Mult2			

- Mult2 (DIVD) is completing; what is waiting for it?



Tomasulo Example Cycle 57

Instruction status:

Instruction	j	k	Issue	Exec		Write	Busy	Address
				Comp	Result			
LD	F6	34+	R2	1	3	4	Load1	No
LD	F2	45+	R3	2	4	5	Load2	No
MULTD	F0	F2	F4	3	15	16	Load3	No
SUBD	F8	F6	F2	4	7	8		
DIVD	F10	F0	F6	5	56	57		
ADDD	F6	F8	F2	6	10	11		

Reservation Stations:

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

Register result status:

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

- Once again: In-order issue, out-of-order execution and out-of-order completion.

Why can Tomasulo overlap iterations of loops?

- Register renaming
 - Multiple iterations use different physical destinations for registers (dynamic loop unrolling).
- Reservation stations
 - Permit instruction issue to advance past integer control flow operations
 - Also buffer old values of registers - totally avoiding the WAR stall
- Other perspective: Tomasulo building data flow dependency graph on the fly



Tomasulo's scheme offers 2 major advantages

- 1. Distribution of the hazard detection logic**
 - distributed reservation stations and the CDB
 - If multiple instructions waiting on single result, & each instruction has other operand, then instructions can be released simultaneously by broadcast on CDB
 - If a centralized register file were used, the units would have to read their results from the registers when register buses are available
- 2. Elimination of stalls for WAW and WAR hazards**

Tomasulo Drawbacks

Complexity

- delays of 360/91, MIPS 10000, Alpha 21264, IBM PPC 620 in CA:AQA 2/e, but not in silicon!
- Many associative stores (CDB) at high speed
- Performance limited by Common Data Bus
 - Each CDB must go to multiple functional units
⇒high capacitance, high wiring density
 - Number of functional units that can complete per cycle limited to one!
 - Multiple CDBs ⇒ more FU logic for parallel assoc stores
- Non-precise interrupts!
 - We will address this later



Outline

- ILP
- Loop unrolling
- Static Branch Prediction
- Dynamic Branch Prediction
- Dynamic Scheduling – Tomasulo's Algorithm
- Reorder Buffer
- CPI less than 1

Speculation to greater ILP

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

Speculation to greater ILP

3 components of HW-based speculation:

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

Adding Speculation to Tomasulo

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

Reorder Buffer (ROB)

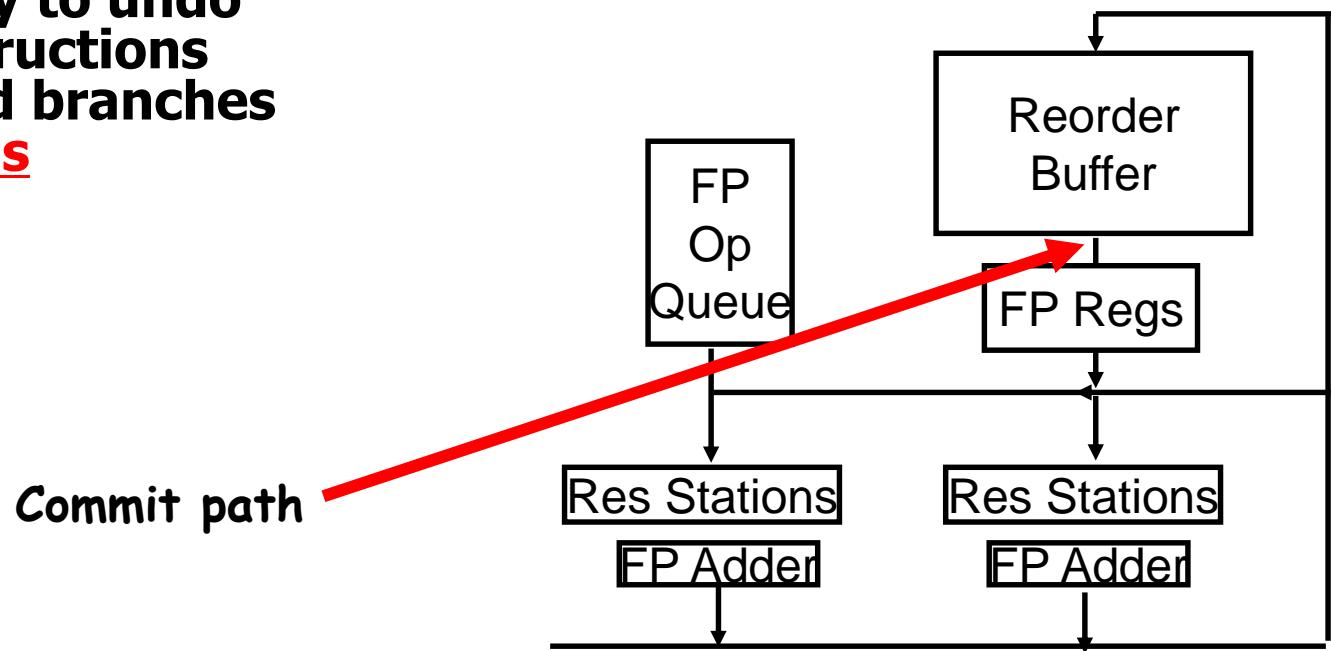
- In Tomasulo's algorithm, once an instruction writes its result, any subsequently issued instructions will find result in the register file
- With speculation, the register file is not updated until the instruction commits
 - we know definitively that the instruction should execute
- Thus, the ROB supplies operands in interval between completion of instruction execution and instruction commit
 - ROB is a source of operands for instructions, just as reservation stations (RS) provide operands in Tomasulo's algorithm
 - ROB extends 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 \Rightarrow more registers like RS
 - Tag results with ROB buffer number instead of reservation station
- Instructions **commit** \Rightarrow values at head of ROB placed in registers
- As a result, easy to undo speculated instructions on mispredicted branches or on exceptions





Recall: 4 Steps of Speculative Tomasulo Algorithm

1. Issue—get instruction from FP Op Queue

If reservation station and reorder buffer slot free, issue instr & send operands & reorder buffer no. for destination (this stage sometimes called "dispatch")

2. Execution—operate on operands (EX)

When both operands ready then execute; if not ready, watch CDB for result; when both in reservation station, execute; checks RAW (sometimes called "issue")

3. Write result—finish execution (WB)

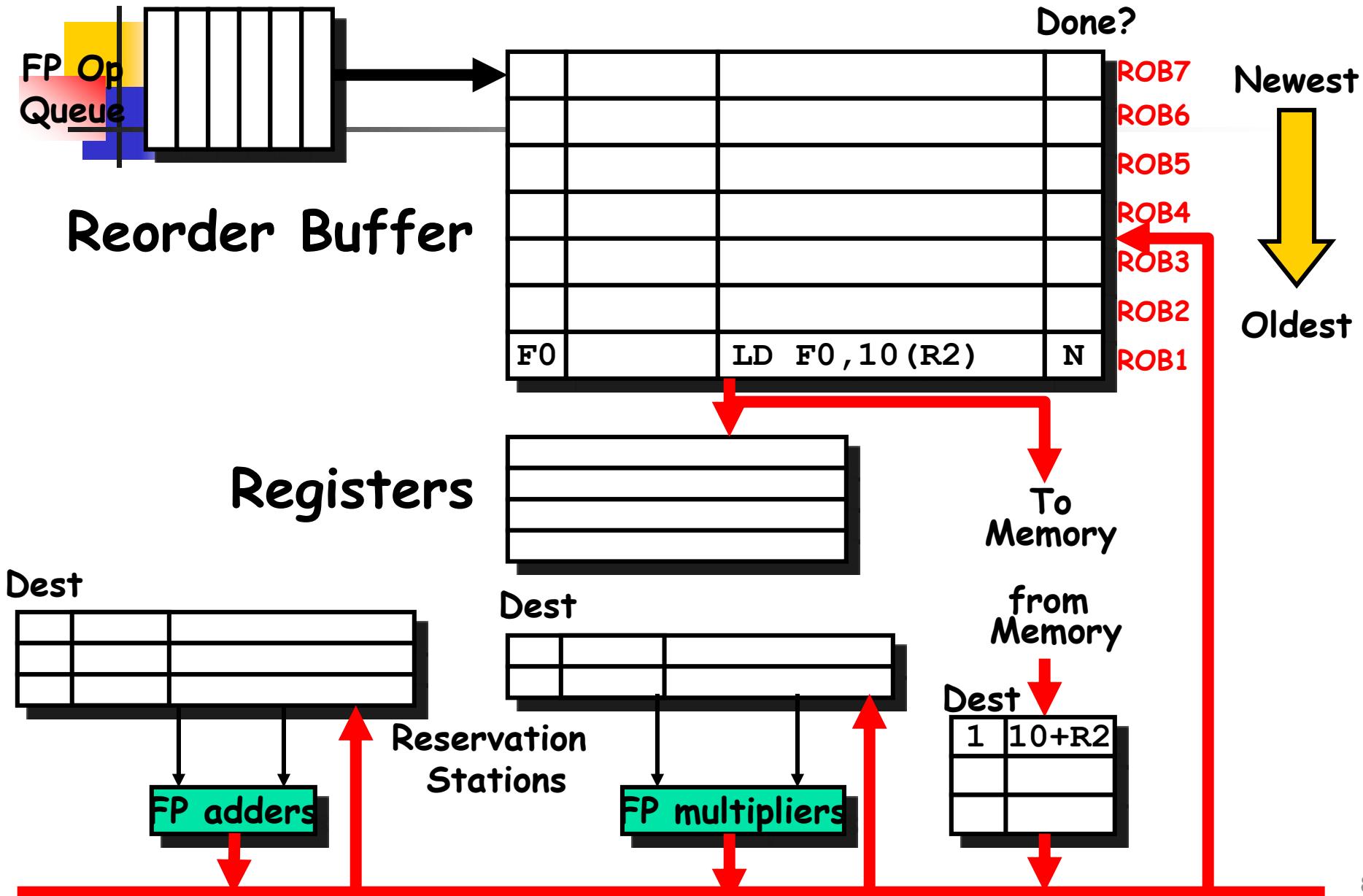
Write on Common Data Bus to all awaiting FUs & reorder buffer; mark reservation station available.

4. Commit—update register with reorder result

When instr. at head of reorder buffer & result present, update register with result (or store to memory) and remove instr from reorder buffer. Mispredicted branch flushes reorder buffer (sometimes called "graduation")



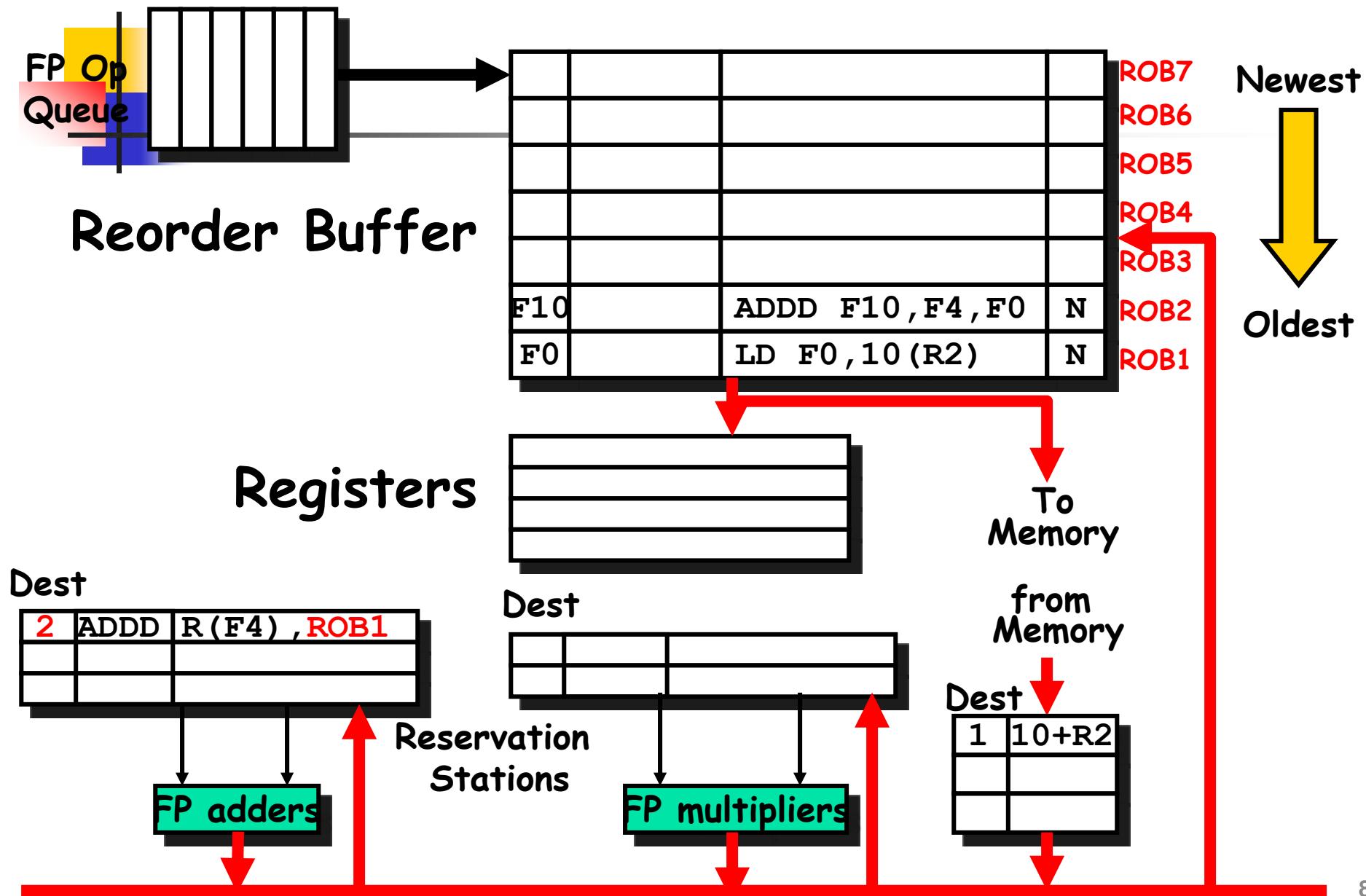
Tomasulo With Reorder buffer:





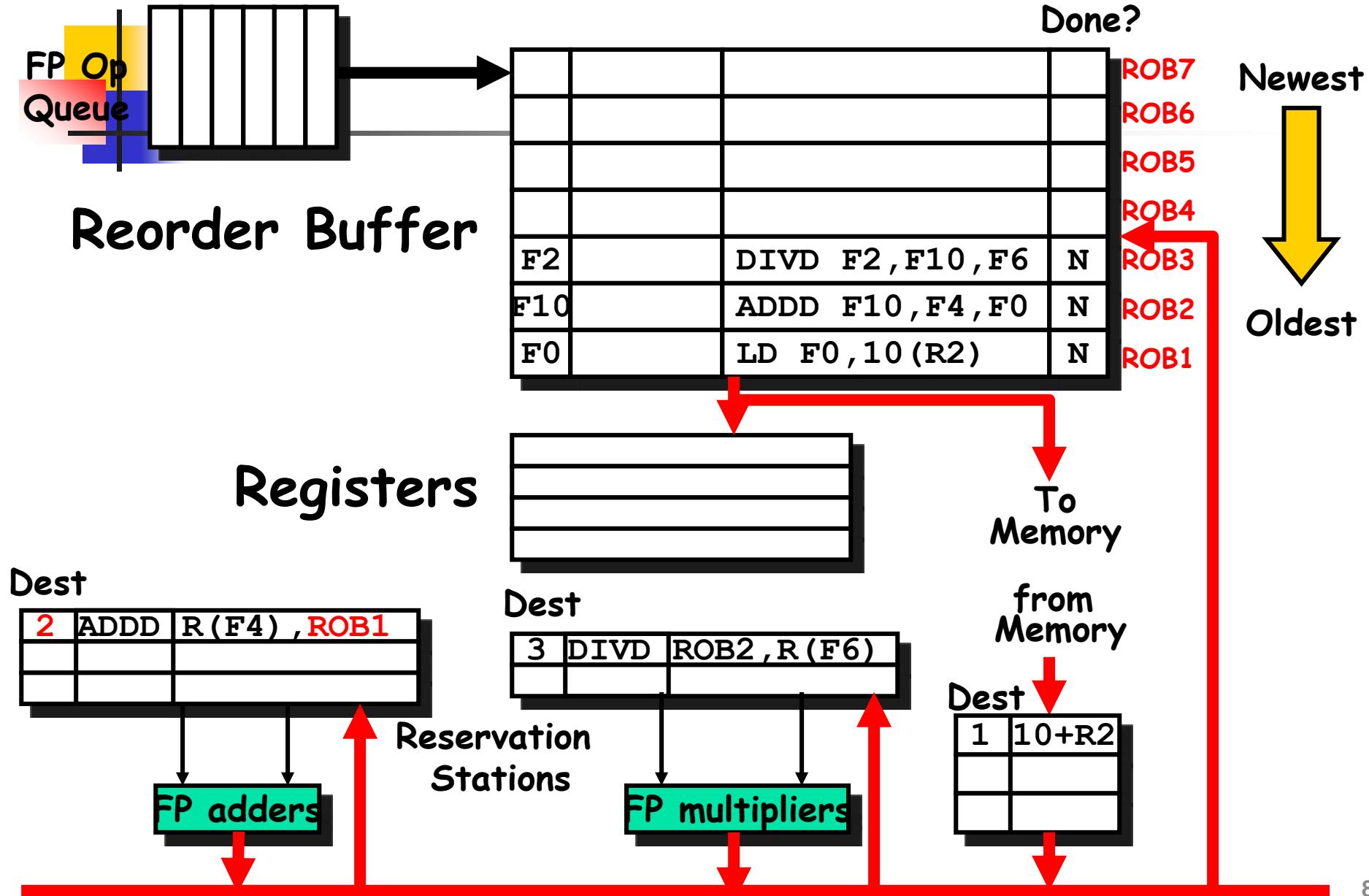
Done?

Tomasulo With Reorder buffer:



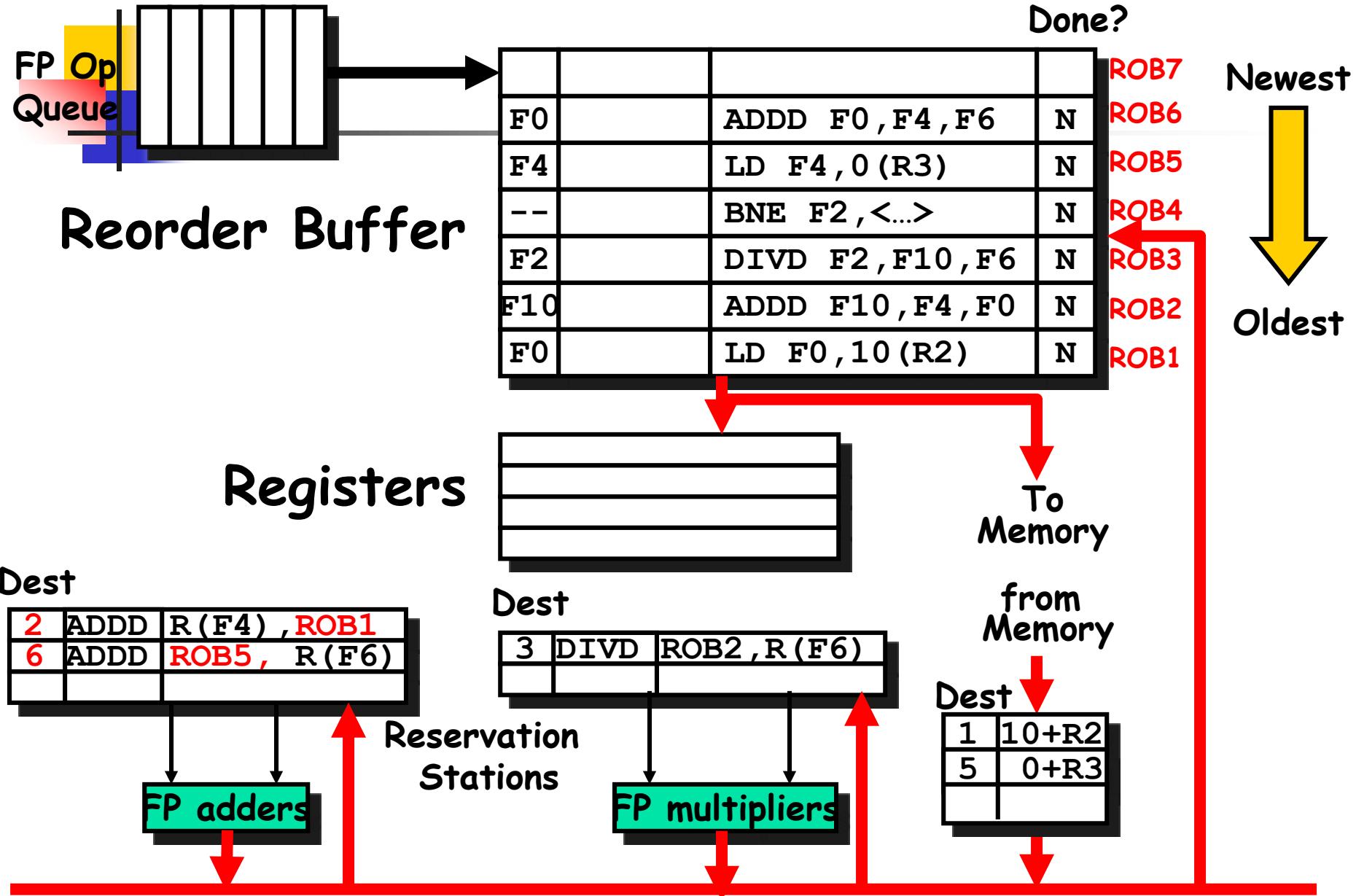


Tomasulo With Reorder buffer:



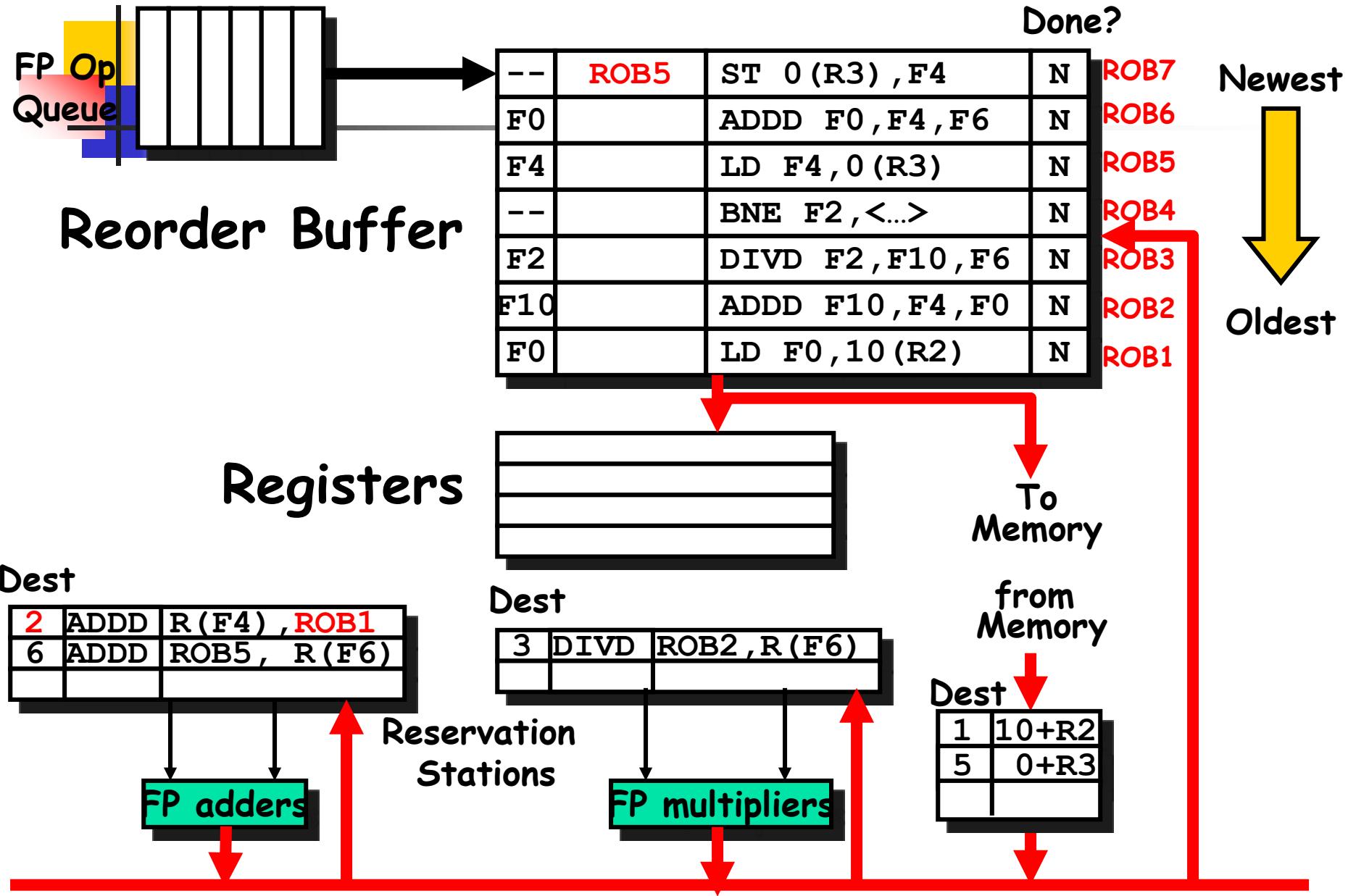


Tomasulo With Reorder buffer:



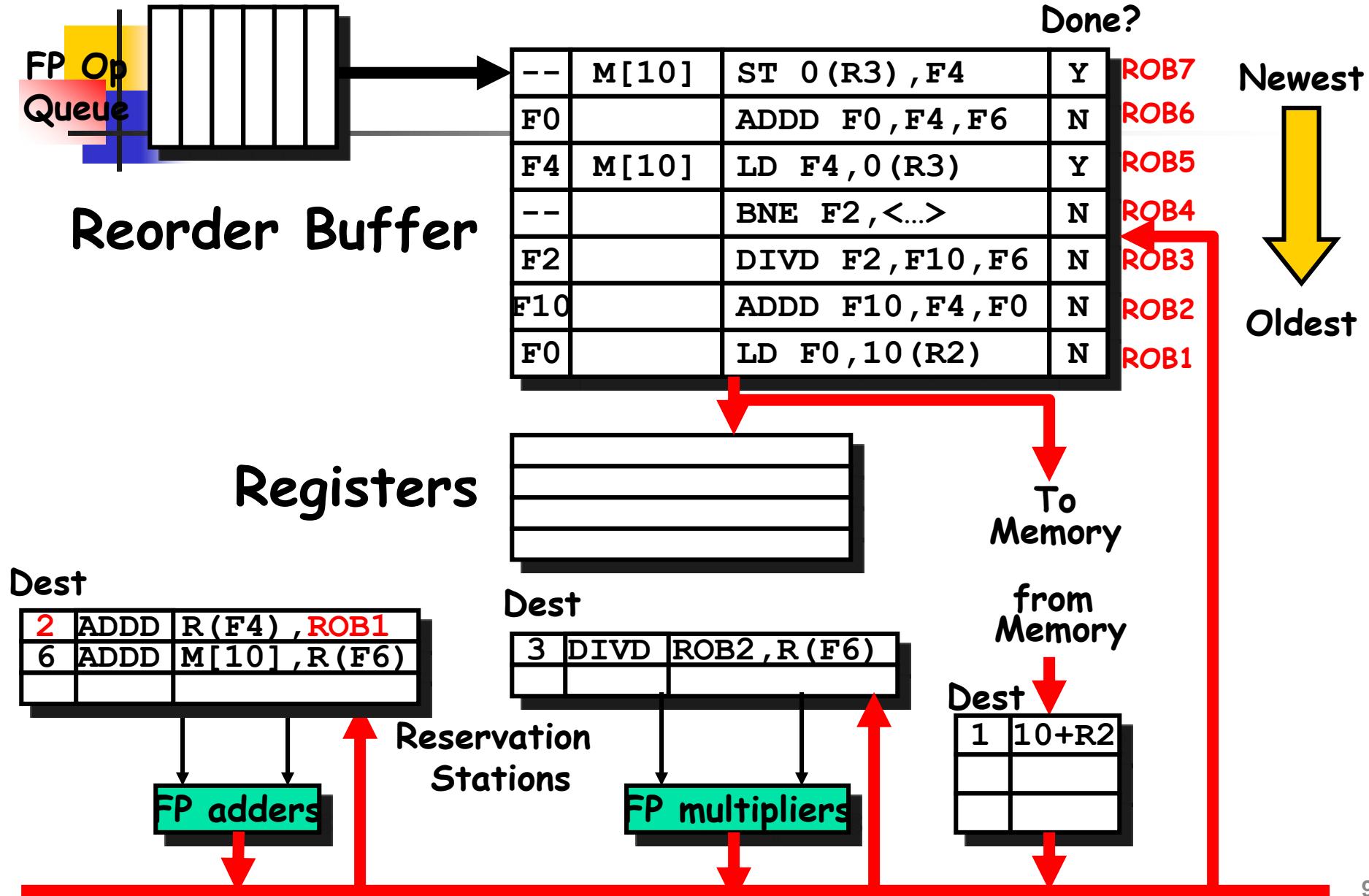


Tomasulo With Reorder buffer:



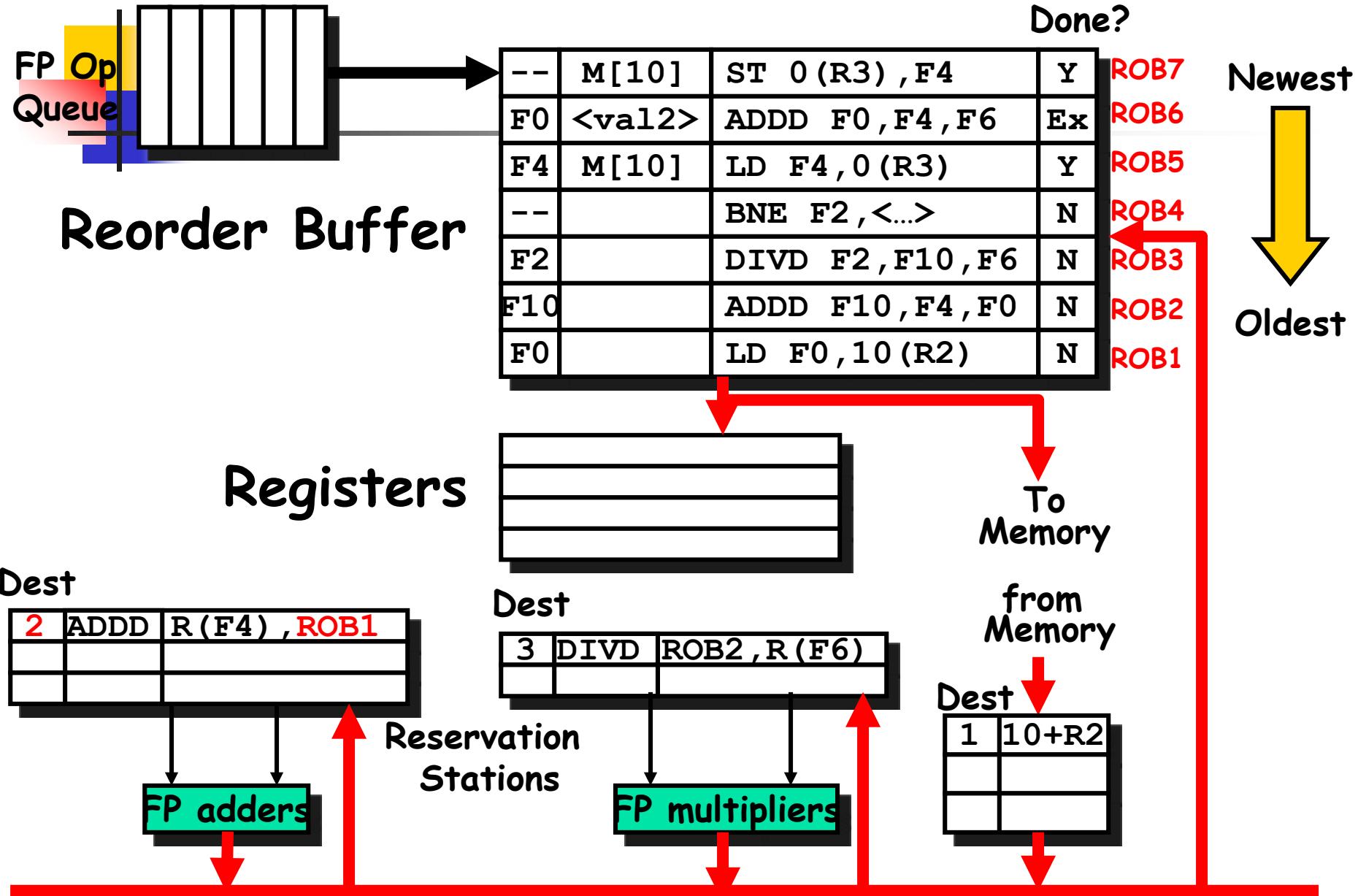


Tomasulo With Reorder buffer:



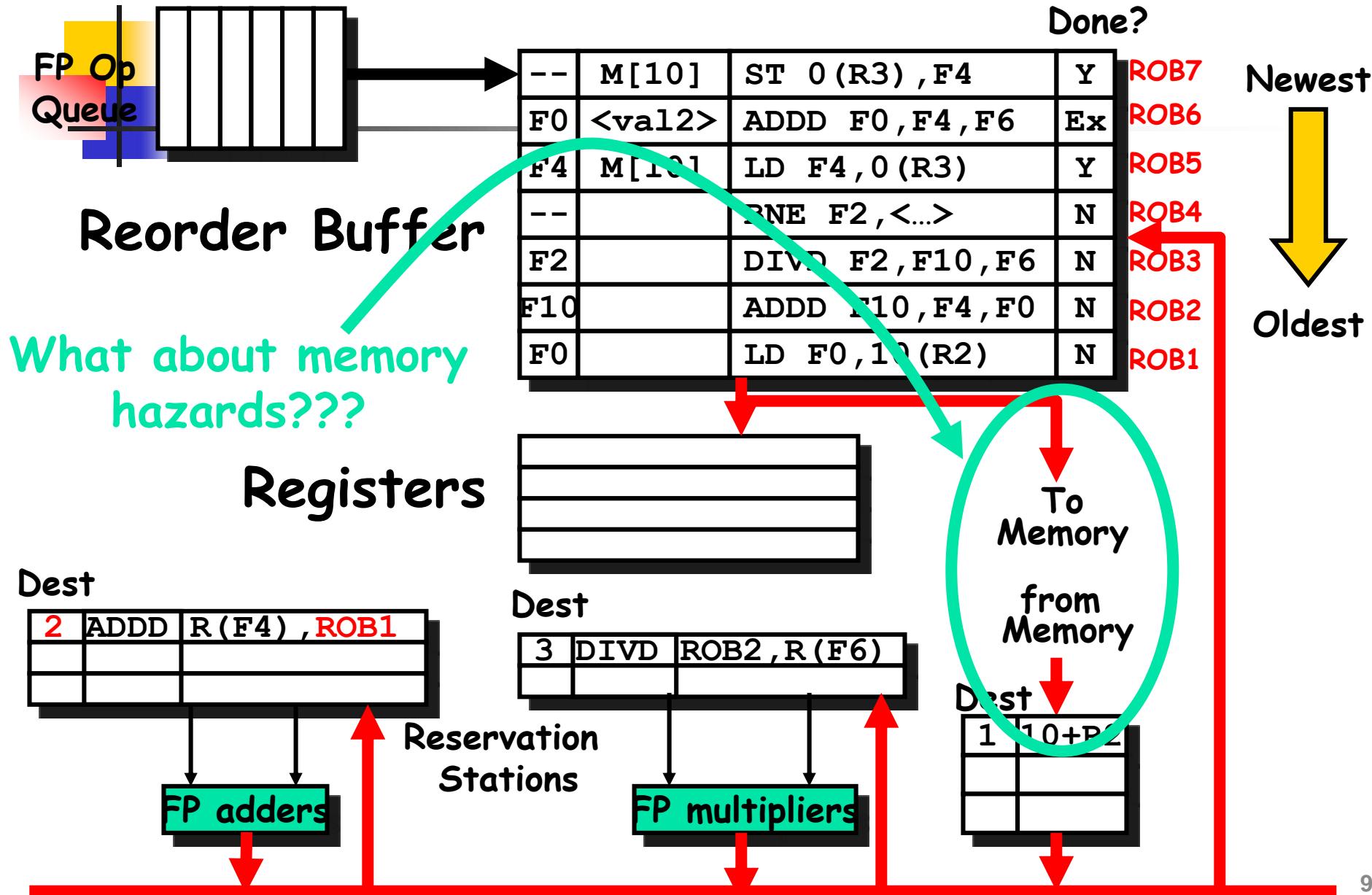


Tomasulo With Reorder buffer:





Tomasulo With Reorder buffer:



Avoiding Memory Hazards

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

Exceptions and Interrupts

IBM 360/91 invented “imprecise interrupts”

- Computer stopped at this PC; its likely close to this address
- Not so popular with programmers
- Also, what about Virtual Memory? (Not in IBM 360)
- Technique for both precise interrupts/exceptions and speculation: in-order completion and in-order commit
 - If we speculate and are wrong, need to back up and restart execution to point at which we predicted incorrectly
 - This is exactly same as need to do with precise exceptions
- Exceptions are handled by not recognizing the exception until instruction that caused it is ready to commit in ROB
 - If a speculated instruction raises an exception, the exception is recorded in the ROB
 - This is why reorder buffers in all new processors



Outline

- ILP
- Loop unrolling
- Static Branch Prediction
- Dynamic Branch Prediction
- Dynamic Scheduling – Tomasulo's Algorithm
- Reorder Buffer
- CPI less than 1

Getting CPI below 1

CPI ≥ 1 if issue only 1 instruction every clock cycle

- **Multiple-issue processors come in 3 flavors:**
 1. statically-scheduled superscalar processors,
 2. dynamically-scheduled superscalar processors, and
 3. VLIW (very long instruction word) processors
- **2 types of superscalar processors issue varying numbers of instructions per clock**
 - use in-order execution if they are statically scheduled, or
 - out-of-order execution if they are dynamically scheduled
- **VLIW processors, in contrast, issue a fixed number of instructions formatted either as one large instruction or as a fixed instruction packet with the parallelism among instructions explicitly indicated by the instruction (Intel/HP Itanium)**

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., 2 integer operations, 2 FP ops, 2 Memory refs, 1 branch
 - 16 to 24 bits per field => 7*16 or 112 bits to 7*24 or 168 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)	L.D to ADD.D: 1 Cycle
2		L.D	F6,-8(R1)	ADD.D to S.D: 2 Cycles
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

14 clock cycles, or 3.5 per iteration

Loop Unrolling in VLIW

<i>Memory reference 1</i>	<i>Memory reference 2</i>	<i>FP operation 1</i>	<i>FP op. 2</i>	<i>Int. op/branch</i>	<i>Clock</i>
L.D F0,0(R1)	L.D F6,-8(R1)				1
L.D F10,-16(R1)	L.D F14,-24(R1)				
		2			
L.D F18,-32(R1)	L.D F22,-40(R1)			ADD.D F4,F0,F2	ADD.D
F8,F6,F2	3				
L.D F26,-48(R1)		ADD.D F12,F10,F2	ADD.D F16,F14,F2		4
		ADD.D F20,F18,F2	ADD.D F24,F22,F2		5
S.D 0(R1),F4	S.D -8(R1),F8	ADD.D F28,F26,F2			6
S.D -16(R1),F12	S.D -24(R1),F16				
7					
S.D -32(R1),F20	S.D -40(R1),F24			DSUBUI R1,R1,#48	
8					
S.D -0(R1),F28				BNEZ R1,LOOP	9

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

Note: Need more registers in VLIW (15 vs. 6 in SS)

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 predict 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)”

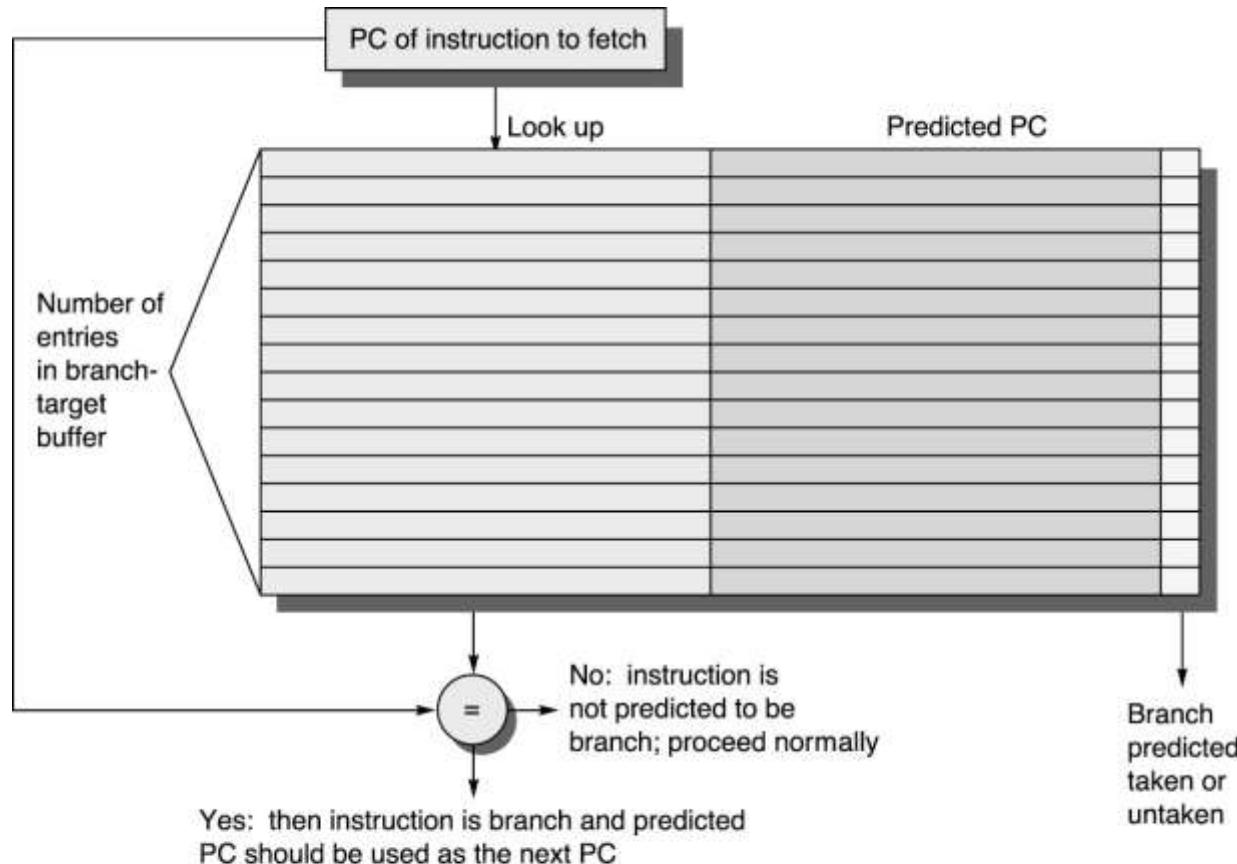
- **IA-64: instruction set architecture**
- **128 64-bit integer regs + 128 82-bit floating point regs**
 - Not separate register files per functional unit as in old VLIW
- **Hardware checks dependencies**
(interlocks => binary compatibility over time)
- **Predicated execution (select 1 out of 64 1-bit flags)**
=> 40% fewer mispredictions?
- **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



Increasing Instruction Fetch Bandwidth

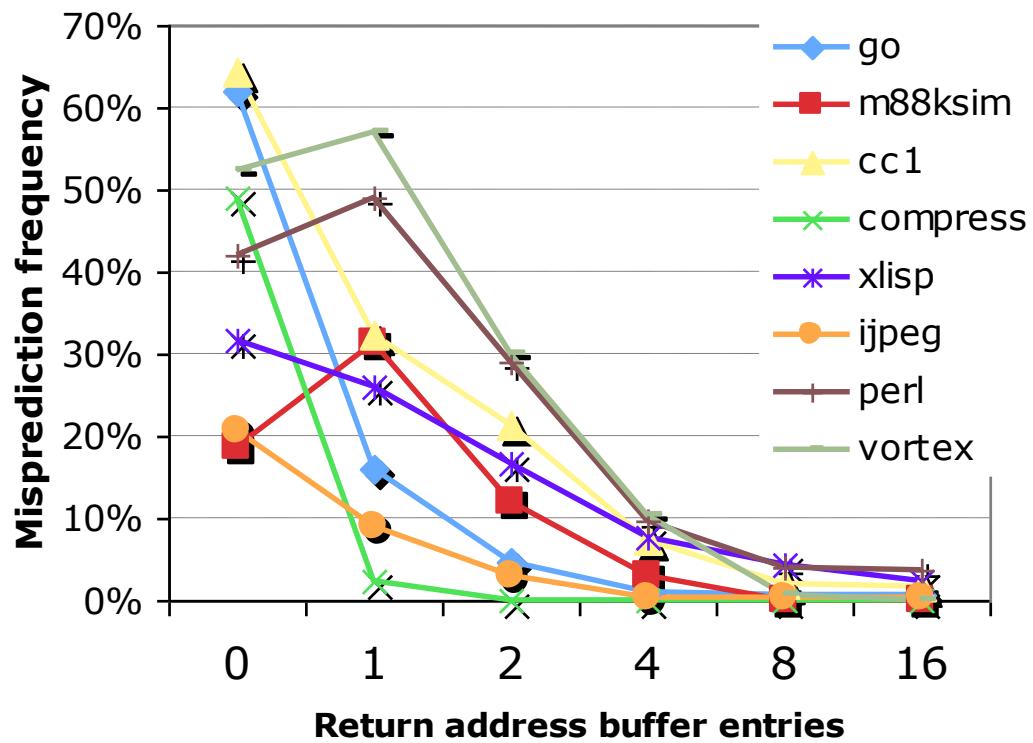
- Predicts next instruction address, sends it out *before* decoding instruction
- PC of branch sent to BTB
- When match is found, Predicted PC is returned
- If branch predicted taken, instruction fetch continues at Predicted PC

Branch Target Buffer (BTB)



IF BW: Return Address Predictor

- Small buffer of return addresses acts as a stack
- Caches most recent return addresses
- Call ⇒ Push a return address on stack
- Return ⇒ Pop an address off stack & predict as new PC



More Instruction Fetch Bandwidth

- **Integrated branch prediction** branch predictor is part of instruction fetch unit and is constantly predicting branches
- **Instruction prefetch** Instruction fetch units prefetch to deliver multiple instruct. per clock, integrating it with branch prediction
- **Instruction memory access and buffering**
Fetching multiple instructions per cycle:
 - May require accessing multiple cache blocks (prefetch to hide cost of crossing cache blocks)
 - Provides buffering, acting as on-demand unit to provide instructions to issue stage as needed and in quantity needed

Speculation: Register Renaming vs. ROB

Alternative to ROB is a larger physical set of registers combined with register renaming

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

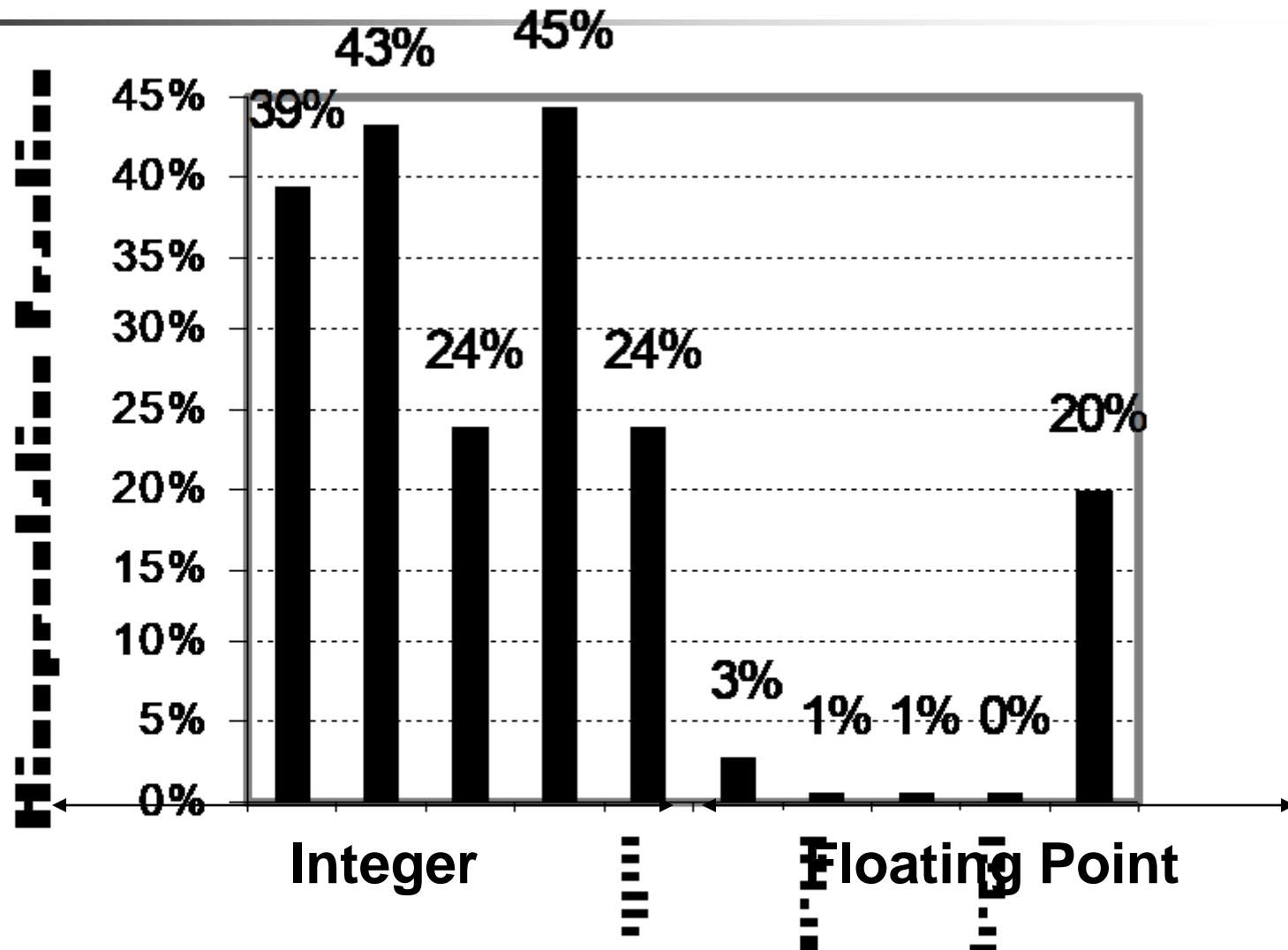
Value Prediction

- Attempts to predict **value** produced by instruction
 - E.g., Loads a value that changes infrequently
- Value prediction is useful only if it significantly increases ILP
 - Focus of research has been on loads; so-so results, no processor uses value prediction
- Related topic is ***address aliasing prediction***
 - RAW for load and store or WAW for 2 stores
- Address alias prediction is both more stable and simpler since need not actually predict the address values, only whether such values conflict
 - Has been used by a few processors



(Mis) Speculation on Pentium 4

- % of micro-ops not used



Perspective

Interest in multiple-issue because wanted to improve performance without affecting uniprocessor programming model

- Taking advantage of ILP is conceptually simple, but design problems are amazingly complex in practice
- Conservative in ideas, just faster clock and bigger
- Processors of last 5 years (Pentium 4, IBM Power 5, AMD Opteron) have the same basic structure and similar sustained issue rates (3 to 4 instructions per clock) as the 1st dynamically scheduled, multiple-issue processors announced in 1995
 - Clocks 10 to 20X faster, caches 4 to 8X bigger, 2 to 4X as many renaming registers, and 2X as many load-store units
⇒ performance 8 to 16X
- Peak v. delivered performance gap increasing



Homework

- Read 3.14