Instruction Level Parallelism

Outline

- ILP
- Compiler techniques to increase ILP
- Loop Unrolling
- Static Branch Prediction
- Dynamic Branch Prediction
- Overcoming Data Hazards with Dynamic Scheduling
- (Start) Tomasulo Algorithm
- Conclusion

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)**

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

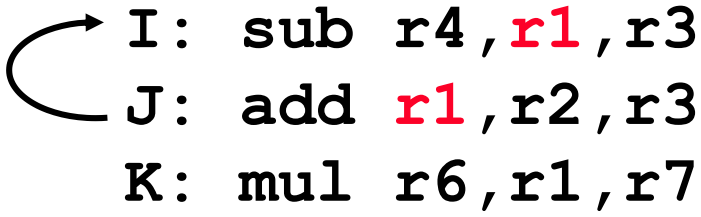
  2. or $Instr_J$ is data dependent on $Instr_K$ which is dependent on $Instr_I$
- If two instructions are data dependent, they cannot execute simultaneously or be completely overlapped
- Data dependence in instruction sequence
  ⇒ data dependence in source code ⇒ effect of original data dependence must 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
  - Dependences 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**

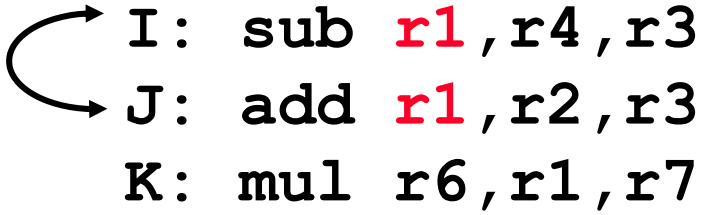


Called an "anti-dependence" by compiler writers. This results from reuse of the name "r1"

- **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.



- 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`?

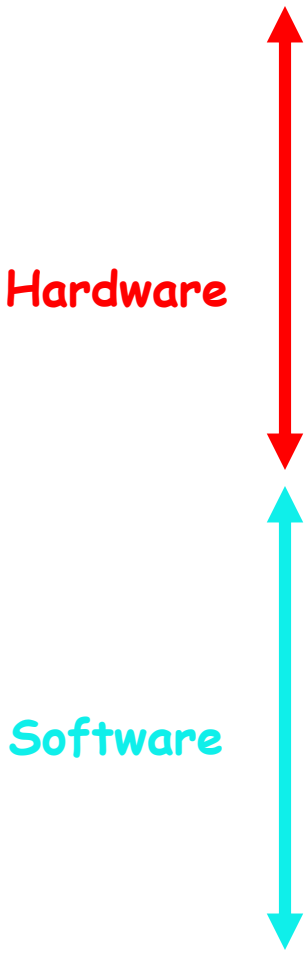
Data Flow

- **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

Ideas to Reduce Stalls



Technique	Reduces
Dynamic scheduling	Data hazard stalls
Dynamic branch prediction	Control stalls
Issuing multiple instructions per cycle	Ideal CPI
Speculation	Data and control stalls
Dynamic memory disambiguation	Data hazard stalls involving memory
Loop unrolling	Control hazard stalls
Basic compiler pipeline scheduling	Data hazard stalls
Compiler dependence analysis	Ideal CPI and data hazard stalls
Software pipelining and trace scheduling	Ideal CPI and data hazard stalls
Compiler speculation	Ideal CPI, data and control stalls

Dynamic Scheduling

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 (next lecture)**

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**

Example on WAR and WAW Hazards

```
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
```

Anti-dependence between ADD.D and SUB.D

WAR hazard on F8

Output dependence between ADD.D and MUL.D WAW hazard on F6

WAR and RAW hazards are caused by out-of-order execution, but can be eliminated with register renaming

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

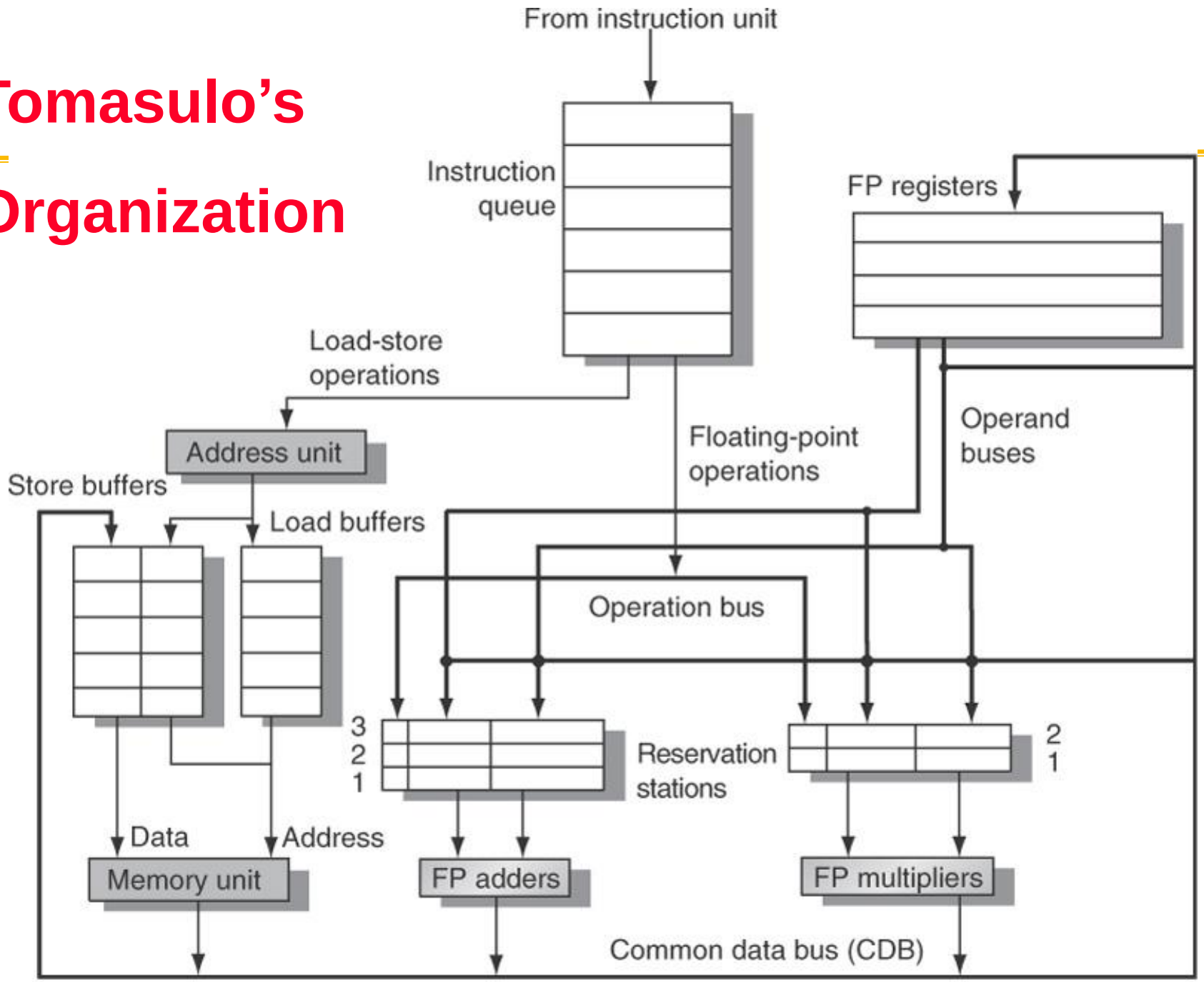
Use temporary registers T1 and T2 to eliminate name dependences

Register renaming can be done statically by the compiler or dynamically by the pipeline

Dynamic Scheduling: Tomasulo

- Fast IBM 360/91 for scientific code
  - Completed in 1967
  - Before cache memories
  - Implemented complex memory system
- Pipelined floating point units
  - FP Adder
  - FP Multiplier (Divide done in multiplier)
- Dynamic scheduling in FP unit (Tomasulo)
- The descendants of Tomasulo are found in
  - Alpha 21264, Pentium 4, AMD Opteron, Power 5, etc

Tomasulo's Organization



Tomasulo Algorithm

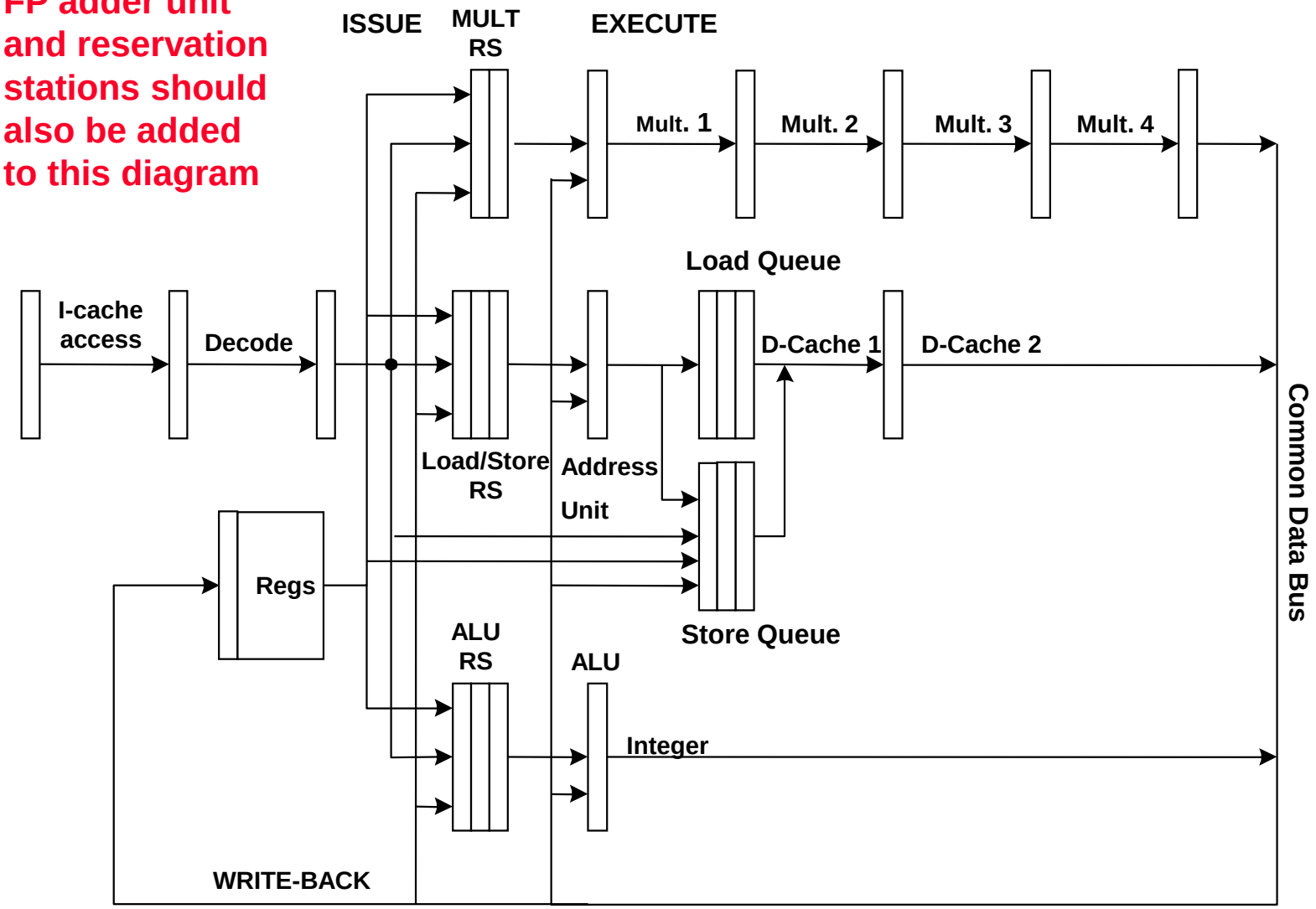
- Buffers & Control distributed with Function Units
  - FU buffers are called "reservation stations" and have pending operands
- Registers in instructions are replaced by values or by pointers to reservation stations, called tags
  - Reservation stations provide renaming to avoid WAR & WAW hazards
- Results are broadcast on the Common Data Bus to all reservation stations, not through registers
  - Avoids RAW hazards by executing an instruction only when its operands are available
- Load and Stores treated as Function Units (FU) with Reservation Stations (RS) as well

Generalized Tomasulo's Organization

- Enhance parallel pipeline architecture
- Apply Tomasulo's algorithm to all pipelined units
  - Not just floating point
  - Biggest difference: handle loads/stores like other instructions
- Use "tags" to identify data values
  - Both tags and register designators can name data
- Reservation Stations (RS) distribute control
  - Set of Reservation Stations per functional unit
  - Tag identifies result of instruction in RS
- Common Data Bus (CDB) broadcasts all results

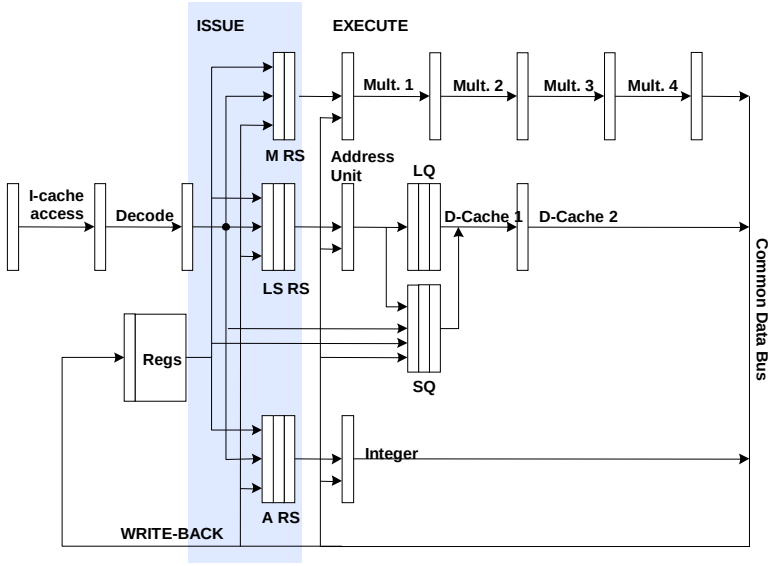
Generalized Tomasulo's Organization

FP adder unit and reservation stations should also be added to this diagram



Three Stages of Tomasulo's Algorithm

1. ISSUE
   - Get next instruction from fetch unit
   - Check for available reservation station
   - If not available, stall due to structural hazard
   - If RS available, *issue*
     - » Copy ready registers to RS
     - » Copy tags for all non-ready registers to RS

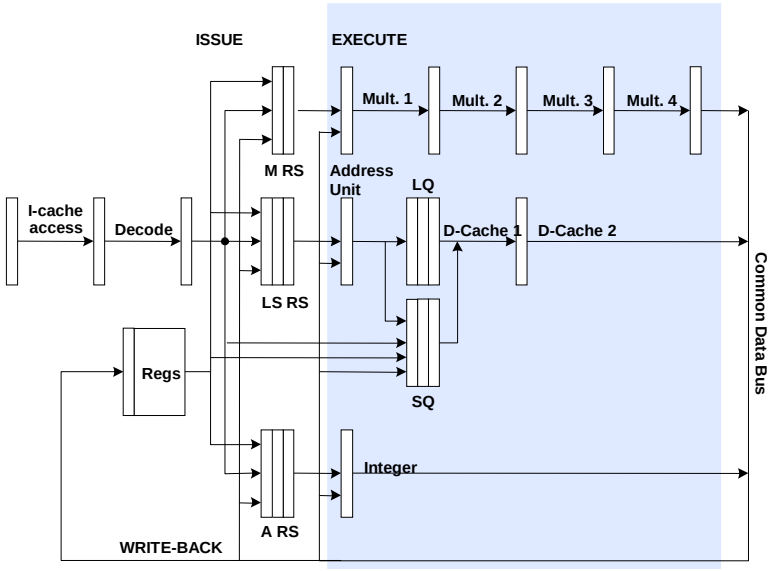


Three Stages of Tomasulo's Algorithm

2. Execute

- If input operands available, issue and begin execution
- If not, monitor CDB for necessary input operands

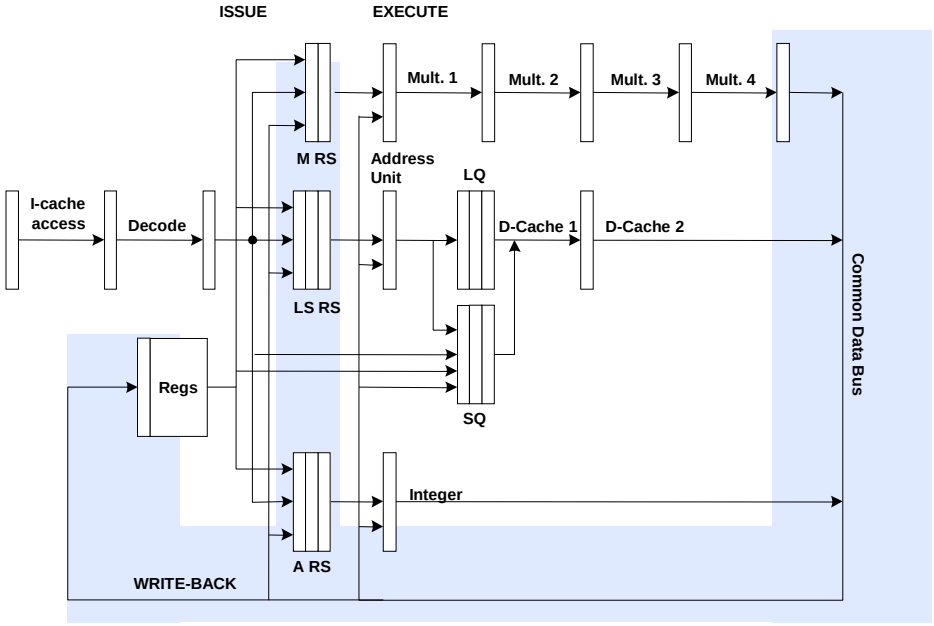
If several instructions become ready for the same functional unit in the same clock cycle, one of them is chosen for execution



Three Stages of Tomasulo's Algorithm

3. Write Back

- If CDB available, write result on CDB
  - » All destinations with matching tags receive data
  - » CDB broadcasts results to all reservation stations
- If not, wait for CDB to become available



Reservation Station Components

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

Vj, Vk: Value of Source operands

- Store buffers has V field, result to be stored

Qj, Qk: Reservation stations producing source registers (value to be written)

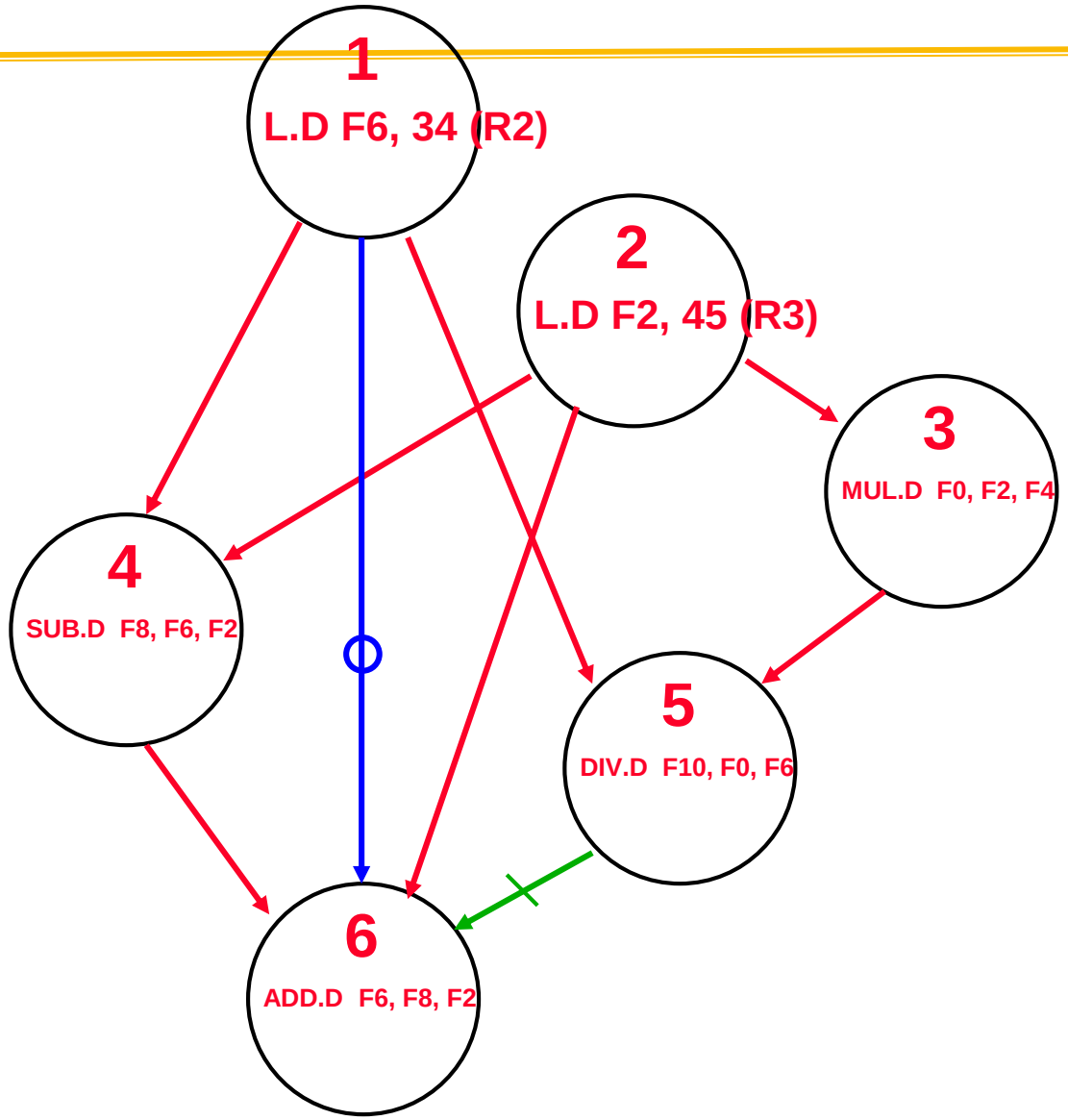
- Note: Qj,Qk=0 => ready
- Store buffers only have Qi for RS producing result

Busy: Indicates reservation station or FU is busy

A Address information for loads or stores. Initially immediate field of instruction then effective address when calculated.

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

Dependency Graph For Example Code



Example Code

1	L.D	F6, 34(R2)
2	L.D	F2, 45(R3)
3	MUL.D	F0, F2, F4
4	SUB.D	F8, F6, F2
5	DIV.D	F10, F0, F6
6	ADD.D	F6, F8, F2

Date Dependence:

(1, 4) (1, 5) (2, 3) (2, 4)

(2, 6) (3, 5) (4, 6)

Output Dependence:

(1, 6)

Anti-dependence:

(5, 6)

The same code used is the scoreboard example

Tomasulo Example

Instruction stream

Instruction status:

Instruction		*j*	*k*	*Issue*	*Exec Comp*	*Write 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
Load1	No	
Load2	No	
Load3	No	

3 Load/Buffers

Reservation Stations:

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

FU count down

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

Register result status:

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

Clock cycle counter

Assume the following latencies: load is 1 clock cycle, add is 2 clock cycles, multiply is 6 clock cycles, and divide is 12 clock cycles.

Tomasulo Example Cycle 1

Instruction status:

Instruction		*j*	*k*	*Issue*	*Exec Comp*	*Write 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 Vj*	*S2 Vk*	*RS Qj*	*RS Qk*
	Add1	No					
	Add2	No					
	Add3	No					
	Mult1	No					
	Mult2	No					

Register result status:

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

Tomasulo Example Cycle 2

Instruction status:

Instruction		*j*	*k*	*Issue*	*Exec Comp*	*Write 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 Vj*	*S2 Vk*	*RS Qj*	*RS 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 Comp	Write Result
LD	F6	34+	R2	1	3	
LD	F2	45+	R3	2		
MULTD	F0	F2	F4	3		
SUBD	F8	F6	F2			
DIVD	F10	F0	F6			
ADDD	F6	F8	F2			

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

Reservation Stations:

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

Register result status:

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

- **Note: registers names are removed (“renamed”) in Reservation Stations; MULT issued**
- **Load1 completing; what is waiting for Load1?**

Tomasulo Example Cycle 4

Instruction status:

Instruction		*j*	*k*	*Issue*	*Exec Comp*	*Write Result*
LD	F6	34+	R2	1	3	4
LD	F2	45+	R3	2	4	
MULTD	F0	F2	F4	3		
SUBD	F8	F6	F2	4		
DIVD	F10	F0	F6			
ADDD	F6	F8	F2			

	Busy	Address
Load1	No	
Load2	Yes	45+R3
Load3	No	

Reservation Stations:

Time	*Name*	*Busy*	*Op*	*S1 Vj*	*S2 Vk*	*RS Qj*	*RS Qk*
	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		*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		
SUBD	F8	F6	F2	4		
DIVD	F10	F0	F6	5		
ADDD	F6	F8	F2			

	Busy	Address
Load1	No	
Load2	No	
Load3	No	

Reservation Stations:

Time	*Name*	*Busy*	*Op*	*S1 Vj*	*S2 Vk*	*RS Qj*	*RS Qk*
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		*F0*	*F2*	*F4*	*F6*	*F8*	*F10*	*F12*	...	*F30*
5	*FU*	Mult1	M(A2)		M(A1)	Add1	Mult2			

- **Timer starts down for Add1, Mult1**

Tomasulo Example Cycle 6

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		
SUBD	F8	F6	F2	4		
DIVD	F10	F0	F6	5		
ADDD	F6	F8	F2	6		

	Busy	Address
Load1	No	
Load2	No	
Load3	No	

Reservation Stations:

Time	*Name*	*Busy*	*Op*	*S1 Vj*	*S2 Vk*	*RS Qj*	*RS Qk*
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		*F0*	*F2*	*F4*	*F6*	*F8*	*F10*	*F12*	...	*F30*
6	*FU*	Mult1	M(A2)		Add2	Add1	Mult2			

- **Issue ADDD here despite name dependency on F6?**

Tomasulo Example Cycle 7

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		
SUBD	F8	F6	F2	4	7	
DIVD	F10	F0	F6	5		
ADDD	F6	F8	F2	6		

	Busy	Address
Load1	No	
Load2	No	
Load3	No	

Reservation Stations:

Time	*Name*	*Busy*	*Op*	*S1 Vj*	*S2 Vk*	*RS Qj*	*RS Qk*
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		*F0*	*F2*	*F4*	*F6*	*F8*	*F10*	*F12*	...	*F30*
7	*FU*	Mult1	M(A2)		Add2	Add1	Mult2			

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

Tomasulo Example Cycle 8

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		
SUBD	F8	F6	F2	4	7	8
DIVD	F10	F0	F6	5		
ADDD	F6	F8	F2	6		

	Busy	Address
Load1	No	
Load2	No	
Load3	No	

Reservation Stations:

Time	*Name*	*Busy*	*Op*	*S1 Vj*	*S2 Vk*	*RS Qj*	*RS Qk*
	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		*F0*	*F2*	*F4*	*F6*	*F8*	*F10*	*F12*	...	*F30*
8	*FU*	Mult1	M(A2)		Add2	(M-M)	Mult2			

Tomasulo Example Cycle 9

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		
SUBD	F8	F6	F2	4	7	8
DIVD	F10	F0	F6	5		
ADDD	F6	F8	F2	6		

	Busy	Address
Load1	No	
Load2	No	
Load3	No	

Reservation Stations:

Time	*Name*	*Busy*	*Op*	*S1 Vj*	*S2 Vk*	*RS Qj*	*RS Qk*
	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		*F0*	*F2*	*F4*	*F6*	*F8*	*F10*	*F12*	...	*F30*
9	*FU*	Mult1	M(A2)		Add2	(M-M)	Mult2			

Tomasulo Example Cycle 10

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		
SUBD	F8	F6	F2	4	7	8
DIVD	F10	F0	F6	5		
ADDD	F6	F8	F2	6	10	

	Busy	Address
Load1	No	
Load2	No	
Load3	No	

Reservation Stations:

Time	*Name*	*Busy*	*Op*	*S1 Vj*	*S2 Vk*	*RS Qj*	*RS Qk*
	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		*F0*	*F2*	*F4*	*F6*	*F8*	*F10*	*F12*	...	*F30*
10	*FU*	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 Comp*	*Write Result*
LD	F6	34+	R2	1	3	4
LD	F2	45+	R3	2	4	5
MULTD	F0	F2	F4	3		
SUBD	F8	F6	F2	4	7	8
DIVD	F10	F0	F6	5		
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					
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		*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		
SUBD	F8	F6	F2	4	7	8
DIVD	F10	F0	F6	5		
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					
3	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*
12	*FU*	Mult1	M(A2)		(M-M+M	(M-M)	Mult2			

Tomasulo Example Cycle 13

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		
SUBD	F8	F6	F2	4	7	8
DIVD	F10	F0	F6	5		
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					
2	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*
13	*FU*	Mult1	M(A2)		(M-M+M	(M-M)	Mult2			

Tomasulo Example Cycle 14

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		
SUBD	F8	F6	F2	4	7	8
DIVD	F10	F0	F6	5		
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					
1	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*
14	*FU*	Mult1	M(A2)		(M-M+M	(M-M)	Mult2			

Tomasulo Example Cycle 15

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	
SUBD	F8	F6	F2	4	7	8
DIVD	F10	F0	F6	5		
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					
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+M	(M-M)	Mult2			

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

Tomasulo Example Cycle 16

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		
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					
40	Mult2	Yes	DIVD	M*F4	M(A1)		

Register result status:

Clock		F0	F2	F4	F6	F8	F10	F12	...	F30
16	FU	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		*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		
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					
1	Mult2	Yes	DIVD	M*F4	M(A1)		

Register result status:

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

Tomasulo Example Cycle 56

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	
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					
0	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+M	(M-M)	Mult2			

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

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)		

Register result status:

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

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

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

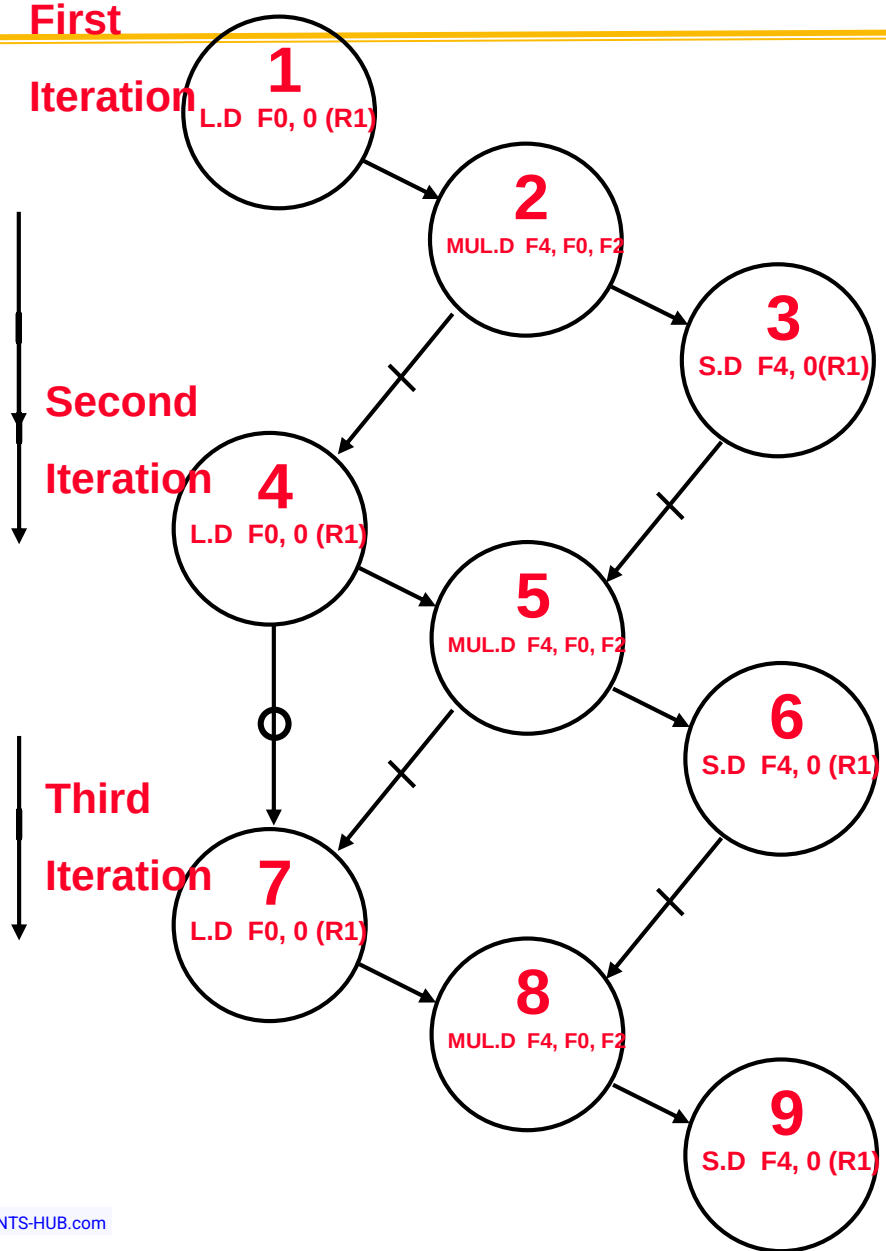
- Complexity
  - delays of 360/91, MIPS 10000, Alpha 21264, IBM PPC 620 in CA:AQA 2/e, but not in silicon!
- Many associative stores (CDB) at high speed
- Performance limited by Common Data Bus
  - Each CDB must go to multiple functional units ⇒high capacitance, high wiring density
  - Number of functional 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

Tomasulo Loop Example

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

- Assume FP Multiply takes 4 execution clock cycles.
- Assume first load takes 8 cycles (possibly due to a cache miss), second load takes 4 cycles (cache hit).
- Assume R1 = 80 initially.
- Assume SUBI only takes one cycle (issue)
- Assume branch resolved in issue stage (no EX or CDB write)
- Assume branch is predicted taken and no branch misprediction.
- No branch delay slot is used in this example.
- Stores take 3 cycles (ex, mem) and do not write on CDB
- We'll go over the execution to complete first two loop iterations.

Tomasulo Loop Example Dependency Graph (First three iterations shown)



Example Code

1	L.D	F0, 0 (R1)
2	MUL.D	F4, F0, F2
3	S.D	F4, 0(R1)
4	L.D	F0, 0(R1)
5	MUL.D	F4, F0, F2
6	S.D	F4, 0(R1)
7	L.D	F0, 0(R1)
8	MUL.D	F4, F0, F2
9	S.D	F4, 0(R1)

Loop maintenance (DADDUI) and branches (BNE) not shown

Name dependencies between iteration 3 instructions and iteration 1 instructions are not shown in graph

Loop Example

Instruction status:

ITER	Instruction	j	k	Issue	Exec Comp	Write Result
1	LD F0	0	R1			
1	MULTD F4	F0	F2			
1	SD F4	0	R1			
2	LD F0	0	R1			
2	MULTD F4	F0	F2			
2	SD F4	0	R1			

Iter-ation Count

	Busy	Addr	Fu
Load1	No		
Load2	No		
Load3	No		
Store1	No		
Store2	No		
Store3	No		

Added Store Buffers

Reservation Stations:

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

Code:

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

Instruction Loop

Register result status

Clock	R1		F0	F2	F4	F6	F8	F10	F12	...	F30
0	80	Fu									

Value of Register used for address, iteration control

Loop Example Cycle 1

Instruction status:

ITER	Instruction	j	k	Issue	Exec Comp	Write Result
1	LD F0	0	R1	1		

	Busy	Addr	Fu
Load1	Yes	80	
Load2	No		
Load3	No		
Store1	No		
Store2	No		
Store3	No		

Reservation Stations:

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

Code:

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

Register result status

Clock	R1		F0	F2	F4	F6	F8	F10	F12	...	F30
1	80	Fu	Load1								

Loop Example Cycle 2

Instruction status:

ITER	Instruction	j	k	Issue	Exec Comp	Write Result
1	LD F0	0	R1	1		
1	MULTD F4	F0	F2	2		

	Busy	Addr	Fu
Load1	Yes	80	
Load2	No		
Load3	No		
Store1	No		
Store2	No		
Store3	No		

Reservation Stations:

Time	Name	Busy	Op	Vj	S1 Vk	S2 Qj	RS Qk
	Add1	No					
	Add2	No					
	Add3	No					
	Mult1	Yes	Multd		R(F2)	Load1	
	Mult2	No					

Code:

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

Register result status

Clock	R1		F0	F2	F4	F6	F8	F10	F12	...	F30
2	80	Fu	Load1		Mult1						

Loop Example Cycle 3

Instruction status:

ITER	Instruction		*j*	*k*	*Issue*	*Exec Comp*	*Write Result*
1	LD	F0	0	R1	1		
1	MULTD	F4	F0	F2	2		
1	SD	F4	0	R1	3		

	Busy	*Addr*	*Fu*
Load1	Yes	80	
Load2	No		
Load3	No		
Store1	Yes	80	Mult1
Store2	No		
Store3	No		

Reservation Stations:

Time	*Name*	*Busy*	*Op*	*S1 Vj*	*S2 Vk*	*RS Qj*	*RS Qk*
	Add1	No					
	Add2	No					
	Add3	No					
	Mult1	Yes	Multd		R(F2)	Load1	
	Mult2	No					

Code:

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

Register result status

Clock	**R1**		*F0*	*F2*	*F4*	*F6*	*F8*	*F10*	*F12*	...	*F30*
3	**80**	*Fu*	Load1		Mult1						

- **Implicit renaming sets up data flow graph**

Loop Example Cycle 4

Instruction status:

ITER	Instruction		j	k	Issue	Exec Comp	Write Result
1	LD	F0	0	R1	1		
1	MULTD	F4	F0	F2	2		
1	SD	F4	0	R1	3		

	Busy	Addr	Fu
Load1	Yes	80	
Load2	No		
Load3	No		
Store1	Yes	80	Mult1
Store2	No		
Store3	No		

Reservation Stations:

Time	Name	Busy	Op	Vj	S1 Vk	S2 Qj	RS Qk
	Add1	No					
	Add2	No					
	Add3	No					
	Mult1	Yes	Multd		R(F2)	Load1	
	Mult2	No					

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

Register result status

Clock	R1		F0	F2	F4	F6	F8	F10	F12	...	F30
4	80	Fu	Load1		Mult1						

- **Dispatching SUBI Instruction (not in FP queue)**

Loop Example Cycle 5

Instruction status:

ITER	Instruction	*j*	*k*	*Issue*	*Comp*	*Result*
1	LD F0	0	R1	1		
1	MULTD F4	F0	F2	2		
1	SD F4	0	R1	3		

Exec Write

	Busy	*Addr*	*Fu*
Load1	Yes	80	
Load2	No		
Load3	No		
Store1	Yes	80	Mult1
Store2	No		
Store3	No		

Reservation Stations:

Time	*Name*	*Busy*	*Op*	*Vj*	*S1* *Vk*	*S2* *Qj*	*RS* *Qk*
	Add1	No					
	Add2	No					
	Add3	No					
	Mult1	Yes	Multd		R(F2)	Load1	
	Mult2	No					

Code:

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

Register result status

Clock	R1		*F0*	*F2*	*F4*	*F6*	*F8*	*F10*	*F12*	...	*F30*
5	72	*Fu*	Load1		Mult1						

- **And, BNEZ instruction (not in FP queue)**

Loop Example Cycle 6

Instruction status:

ITER	Instruction	j	k	Issue	Exec Comp	Write Result
1	LD F0	0	R1	1		
1	MULTD F4	F0	F2	2		
1	SD F4	0	R1	3		
2	LD F0	0	R1	6		

	Busy	Addr	Fu
Load1	Yes	80	
Load2	Yes	72	
Load3	No		
Store1	Yes	80	Mult1
Store2	No		
Store3	No		

Reservation Stations:

Time	Name	Busy	Op	Vj	S1 Vk	S2 Qj	RS Qk
	Add1	No					
	Add2	No					
	Add3	No					
	Mult1	Yes	Multd		R(F2)	Load1	
	Mult2	No					

Code:

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

Register result status

Clock	R1		F0	F2	F4	F6	F8	F10	F12	...	F30
6	72	Fu	Load2		Mult1						

- **Notice that F0 never sees Load from location 80**

Loop Example Cycle 7

Instruction status:

ITER	Instruction		j	k	Issue	Exec Comp	Write Result
1	LD	F0	0	R1	1		
1	MULTD	F4	F0	F2	2		
1	SD	F4	0	R1	3		
2	LD	F0	0	R1	6		
2	MULTD	F4	F0	F2	7		

	Busy	Addr	Fu
Load1	Yes	80	
Load2	Yes	72	
Load3	No		
Store1	Yes	80	Mult1
Store2	No		
Store3	No		

Reservation Stations:

Time	Name	Busy	Op	S1 Vj	S2 Vk	RS Qj	RS Qk
	Add1	No					
	Add2	No					
	Add3	No					
	Mult1	Yes	Multd		R(F2)	Load1	
	Mult2	Yes	Multd		R(F2)	Load2	

Code:

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

Register result status

Clock	R1		F0	F2	F4	F6	F8	F10	F12	...	F30
7	72	Fu	Load2		Mult2						

- **Register file completely detached from computation**
- **First and Second iteration completely overlapped**

Loop Example Cycle 8

Instruction status:

ITER	Instruction		j	k	Issue	Exec Comp	Write Result
1	LD	F0	0	R1	1		
1	MULTD	F4	F0	F2	2		
1	SD	F4	0	R1	3		
2	LD	F0	0	R1	6		
2	MULTD	F4	F0	F2	7		
2	SD	F4	0	R1	8		

	Busy	Addr	Fu
Load1	Yes	80	
Load2	Yes	72	
Load3	No		
Store1	Yes	80	Mult1
Store2	Yes	72	Mult2
Store3	No		

Reservation Stations:

Time	Name	Busy	Op	Vj	S1 Vk	S2 Qj	RS Qk
	Add1	No					
	Add2	No					
	Add3	No					
	Mult1	Yes	Multd		R(F2)	Load1	
	Mult2	Yes	Multd		R(F2)	Load2	

Code:

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

Register result status

Clock	R1		F0	F2	F4	F6	F8	F10	F12	...	F30
8	72	Fu	Load2		Mult2						

Loop Example Cycle 9

Instruction status:

ITER	Instruction		j	k	Issue	Exec Comp	Write Result
1	LD	F0	0	R1	1	9	
1	MULTD	F4	F0	F2	2		
1	SD	F4	0	R1	3		
2	LD	F0	0	R1	6		
2	MULTD	F4	F0	F2	7		
2	SD	F4	0	R1	8		

	Busy	Addr	Fu
Load1	Yes	80	
Load2	Yes	72	
Load3	No		
Store1	Yes	80	Mult1
Store2	Yes	72	Mult2
Store3	No		

Reservation Stations:

Time	Name	Busy	Op	S1 Vj	S2 Vk	RS Qj	RS Qk
	Add1	No					
	Add2	No					
	Add3	No					
	Mult1	Yes	Multd		R(F2)	Load1	
	Mult2	Yes	Multd		R(F2)	Load2	

Code:

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

Register result status

Clock	R1		F0	F2	F4	F6	F8	F10	F12	...	F30
9	72	Fu	Load2		Mult2						

- **Load1 completing: who is waiting?**
- **Note: Dispatching SUBI**

Loop Example Cycle 10

Instruction status:

ITER	Instruction		j	k	Issue	Exec Comp	Write Result
1	LD	F0	0	R1	1	9	10
1	MULTD	F4	F0	F2	2		
1	SD	F4	0	R1	3		
2	LD	F0	0	R1	6	10	
2	MULTD	F4	F0	F2	7		
2	SD	F4	0	R1	8		

	Busy	Addr	Fu
Load1	No		
Load2	Yes	72	
Load3	No		
Store1	Yes	80	Mult1
Store2	Yes	72	Mult2
Store3	No		

Reservation Stations:

Time	Name	Busy	Op	S1 Vj	S1 Vk	S2 Qj	S2 Qk	RS
	Add1	No						
	Add2	No						
	Add3	No						
4	Mult1	Yes	Multd	M[80]	R(F2)			
	Mult2	Yes	Multd		R(F2)	Load2		

Code:

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

Register result status

Clock	R1		F0	F2	F4	F6	F8	F10	F12	...	F30
10	64	Fu	Load2		Mult2						

- **Load2 completing: who is waiting?**
- **Note: Dispatching BNEZ**

Loop Example Cycle 11

Instruction status:

ITER	Instruction		j	k	Issue	Exec Comp	Write Result
1	LD	F0	0	R1	1	9	10
1	MULTD	F4	F0	F2	2		
1	SD	F4	0	R1	3		
2	LD	F0	0	R1	6	10	11
2	MULTD	F4	F0	F2	7		
2	SD	F4	0	R1	8		

	Busy	Addr	Fu
Load1	No		
Load2	No		
Load3	Yes	64	
Store1	Yes	80	Mult1
Store2	Yes	72	Mult2
Store3	No		

Reservation Stations:

Time	Name	Busy	Op	S1 Vj	S2 Vk	RS Qj	RS Qk
	Add1	No					
	Add2	No					
	Add3	No					
3	Mult1	Yes	Multd	M[80]	R(F2)		
4	Mult2	Yes	Mult	M[72]	R(F2)		

Code:

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

Register result status

Clock	R1		F0	F2	F4	F6	F8	F10	F12	...	F30
11	64	Fu	Load3		Mult2						

- **Next load in sequence**

Loop Example Cycle 12

Instruction status:

ITER	Instruction		j	k	Issue	Exec Comp	Write Result
1	LD	F0	0	R1	1	9	10
1	MULTD	F4	F0	F2	2		
1	SD	F4	0	R1	3		
2	LD	F0	0	R1	6	10	11
2	MULTD	F4	F0	F2	7		
2	SD	F4	0	R1	8		

	Busy	Addr	Fu
Load1	No		
Load2	No		
Load3	Yes	64	
Store1	Yes	80	Mult1
Store2	Yes	72	Mult2
Store3	No		

Reservation Stations:

Time	Name	Busy	Op	S1 Vj	S2 Vk	RS Qj	RS Qk
	Add1	No					
	Add2	No					
	Add3	No					
2	Mult1	Yes	Multd	M[80]	R(F2)		
3	Mult2	Yes	Multd	M[72]	R(F2)		

Code:

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

Register result status

Clock	R1		F0	F2	F4	F6	F8	F10	F12	...	F30
12	64	Fu	Load3		Mult2						

- **Why not issue third multiply?**

Loop Example Cycle 13

Instruction status:

ITER	Instruction		j	k	Issue	Exec Comp	Write Result
1	LD	F0	0	R1	1	9	10
1	MULTD	F4	F0	F2	2		
1	SD	F4	0	R1	3		
2	LD	F0	0	R1	6	10	11
2	MULTD	F4	F0	F2	7		
2	SD	F4	0	R1	8		

	Busy	Addr	Fu
Load1	No		
Load2	No		
Load3	Yes	64	
Store1	Yes	80	Mult1
Store2	Yes	72	Mult2
Store3	No		

Reservation Stations:

Time	Name	Busy	Op	S1 Vj	S2 Vk	RS Qj	RS Qk
	Add1	No					
	Add2	No					
	Add3	No					
1	Mult1	Yes	Multd	M[80]	R(F2)		
2	Mult2	Yes	Multd	M[72]	R(F2)		

Code:

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

Register result status

Clock	R1		F0	F2	F4	F6	F8	F10	F12	...	F30
13	64	Fu	Load3		Mult2						

- **Why not issue third store?**

Loop Example Cycle 14

Instruction status:

ITER	Instruction		j	k	Issue	Exec Comp	Write Result
1	LD	F0	0	R1	1	9	10
1	MULTD	F4	F0	F2	2	14	
1	SD	F4	0	R1	3		
2	LD	F0	0	R1	6	10	11
2	MULTD	F4	F0	F2	7		
2	SD	F4	0	R1	8		

	Busy	Addr	Fu
Load1	No		
Load2	No		
Load3	Yes	64	
Store1	Yes	80	Mult1
Store2	Yes	72	Mult2
Store3	No		

Reservation Stations:

Time	Name	Busy	Op	S1 Vj	S2 Vk	RS Qj	RS Qk
	Add1	No					
	Add2	No					
	Add3	No					
0	Mult1	Yes	Multd	M[80]	R(F2)		
1	Mult2	Yes	Multd	M[72]	R(F2)		

Code:

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

Register result status

Clock	R1		F0	F2	F4	F6	F8	F10	F12	...	F30
14	64	Fu	Load3		Mult2						

- **Mult1 completing. Who is waiting?**

Loop Example Cycle 15

Instruction status:

ITER	Instruction		j	k	Issue	Exec Comp	Write Result
1	LD	F0	0	R1	1	9	10
1	MULTD	F4	F0	F2	2	14	15
1	SD	F4	0	R1	3		
2	LD	F0	0	R1	6	10	11
2	MULTD	F4	F0	F2	7	15	
2	SD	F4	0	R1	8		

	Busy	Addr	Fu
Load1	No		
Load2	No		
Load3	Yes	64	
Store1	Yes	80	[80]*R2
Store2	Yes	72	Mult2
Store3	No		

Reservation Stations:

Time	Name	Busy	Op	S1 Vj	S1 Vk	S2 Qj	S2 Qk	RS
	Add1	No						
	Add2	No						
	Add3	No						
	Mult1	No						
0	Mult2	Yes	Multd	M[72]		R(F2)		

Code:

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

Register result status

Clock	R1		F0	F2	F4	F6	F8	F10	F12	...	F30
15	64	Fu	Load3		Mult2						

- **Mult2 completing. Who is waiting?**

Loop Example Cycle 16

Instruction status:

ITER	Instruction		j	k	Issue	Exec Comp	Write Result
1	LD	F0	0	R1	1	9	10
1	MULTD	F4	F0	F2	2	14	15
1	SD	F4	0	R1	3		
2	LD	F0	0	R1	6	10	11
2	MULTD	F4	F0	F2	7	15	16
2	SD	F4	0	R1	8		

	Busy	Addr	Fu
Load1	No		
Load2	No		
Load3	Yes	64	
Store1	Yes	80	[80]*R2
Store2	Yes	72	[72]*R2
Store3	No		

Reservation Stations:

Time	Name	Busy	Op	S1 Vj	S2 Vk	RS Qj	RS Qk
	Add1	No					
	Add2	No					
	Add3	No					
4	Mult1	Yes	Multd		R(F2)	Load3	
	Mult2	No					

Code:

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

Register result status

Clock	R1		F0	F2	F4	F6	F8	F10	F12	...	F30
16	64	Fu	Load3		Mult1						

Loop Example Cycle 17

Instruction status:

ITER	Instruction		j	k	Issue	Exec Comp	Write Result
1	LD	F0	0	R1	1	9	10
1	MULTD	F4	F0	F2	2	14	15
1	SD	F4	0	R1	3		
2	LD	F0	0	R1	6	10	11
2	MULTD	F4	F0	F2	7	15	16
2	SD	F4	0	R1	8		

	Busy	Addr	Fu
Load1	No		
Load2	No		
Load3	Yes	64	
Store1	Yes	80	[80]*R2
Store2	Yes	72	[72]*R2
Store3	Yes	64	Mult1

Reservation Stations:

Time	Name	Busy	Op	S1 Vj	S2 Vk	RS Qj	RS Qk
	Add1	No					
	Add2	No					
	Add3	No					
	Mult1	Yes	Multd		R(F2)	Load3	
	Mult2	No					

Code:

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

Register result status

Clock	R1		F0	F2	F4	F6	F8	F10	F12	...	F30
17	64	Fu	Load3		Mult1						

Loop Example Cycle 18

Instruction status:

ITER	Instruction		j	k	Issue	Exec Comp	Write Result
1	LD	F0	0	R1	1	9	10
1	MULTD	F4	F0	F2	2	14	15
1	SD	F4	0	R1	3	18	
2	LD	F0	0	R1	6	10	11
2	MULTD	F4	F0	F2	7	15	16
2	SD	F4	0	R1	8		

	Busy	Addr	Fu
Load1	No		
Load2	No		
Load3	Yes	64	
Store1	Yes	80	[80]*R2
Store2	Yes	72	[72]*R2
Store3	Yes	64	Mult1

Reservation Stations:

Time	Name	Busy	Op	S1 Vj	S2 Vk	RS Qj	Qk
	Add1	No					
	Add2	No					
	Add3	No					
	Mult1	Yes	Multd	R(F2)	Load3		
	Mult2	No					

Code:

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

Register result status

Clock	R1		F0	F2	F4	F6	F8	F10	F12	...	F30
18	64	Fu	Load3		Mult1						

Loop Example Cycle 19

Instruction status:

ITER	Instruction		j	k	Issue	Exec Comp	Write Result
1	LD	F0	0	R1	1	9	10
1	MULTD	F4	F0	F2	2	14	15
1	SD	F4	0	R1	3	18	19
2	LD	F0	0	R1	6	10	11
2	MULTD	F4	F0	F2	7	15	16
2	SD	F4	0	R1	8	19	

	Busy	Addr	Fu
Load1	No		
Load2	No		
Load3	Yes	64	
Store1	No		
Store2	Yes	72	[72]*R2
Store3	Yes	64	Mult1

Reservation Stations:

Time	Name	Busy	Op	Vj	S1 Vk	S2 Qj	RS Qk
	Add1	No					
	Add2	No					
	Add3	No					
	Mult1	Yes	Multd		R(F2)	Load3	
	Mult2	No					

Code:

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

Register result status

Clock	R1		F0	F2	F4	F6	F8	F10	F12	...	F30
19	56	Fu	Load3		Mult1						

Loop Example Cycle 20

Instruction status:

ITER	Instruction		j	k	Issue	Exec Comp	Write Result
1	LD	F0	0	R1	1	9	10
1	MULTD	F4	F0	F2	2	14	15
1	SD	F4	0	R1	3	18	19
2	LD	F0	0	R1	6	10	11
2	MULTD	F4	F0	F2	7	15	16
2	SD	F4	0	R1	8	19	20

	Busy	Addr	Fu
Load1	Yes	56	
Load2	No		
Load3	Yes	64	
Store1	No		
Store2	No		
Store3	Yes	64	Mult1

Reservation Stations:

Time	Name	Busy	Op	Vj	S1 Vk	S2 Qj	RS Qk
	Add1	No					
	Add2	No					
	Add3	No					
	Mult1	Yes	Multd		R(F2)	Load3	
	Mult2	No					

Code:

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

Register result status

Clock	R1		F0	F2	F4	F6	F8	F10	F12	...	F30
20	56	Fu	Load1		Mult1						

- **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 that we saw in the scoreboard.**
- **Other perspective: Tomasulo building data flow dependency graph on the fly.**

Explicit Register Renaming

- Make use of a *physical* register file that is larger than number of registers specified by ISA
- Keep a translation table:
  - ISA register => physical register mapping
  - When register is written, replace table entry with new register from freelist.
  - Physical register becomes free when not being used by any instructions in progress.
- Pipeline can be exactly like "standard" DLX pipeline
  - IF, ID, EX, etc….
- Advantages:
  - Removes all WAR and WAW hazards
  - Like Tomasulo, good for allowing full out-of-order completion
  - Allows data to be fetched from a single register file
  - Makes speculative execution/precise interrupts easier:
    - » All that needs to be "undone" for precise break point is to undo the table mappings

Explicit Renaming Support Includes:

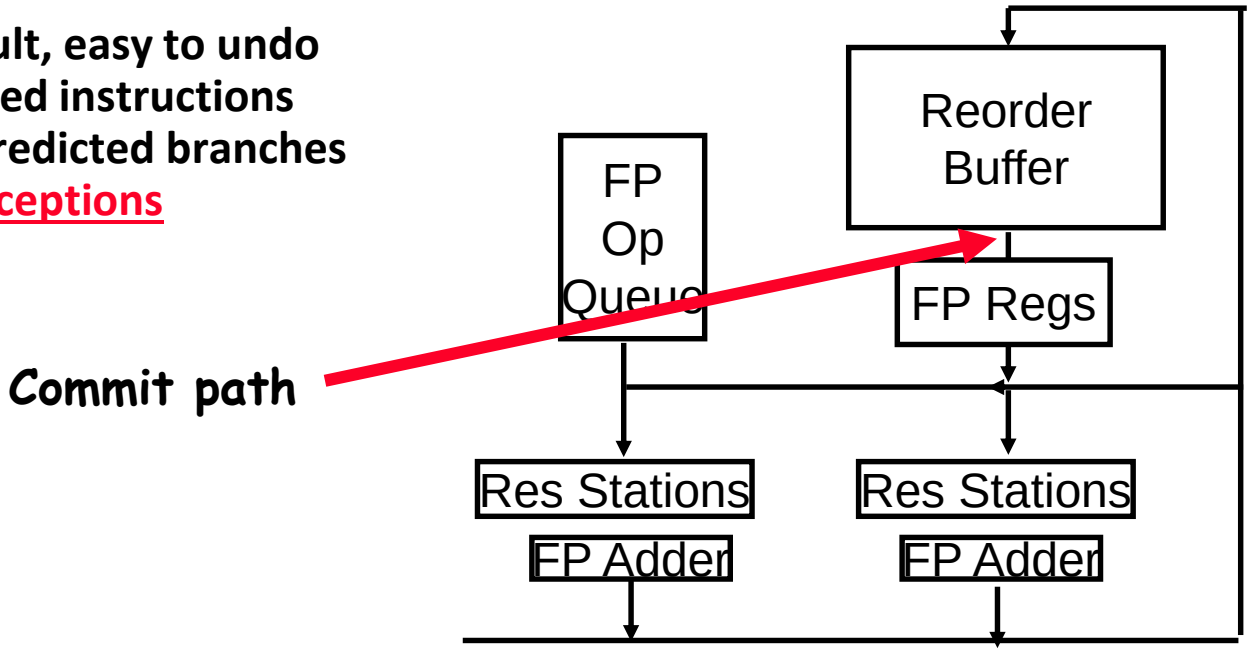
- Rapid access to a table of translations
- A physical register file that has more registers than specified by the ISA
- Ability to figure out which physical registers are free.
  - No free registers ⇒ stall on issue
- Thus, register renaming doesn't require reservation stations. However:
  - Many modern architectures use explicit register renaming + Tomasulo-like reservation stations to control execution.

What about Precise Exceptions/Interrupts?

- **Both Scoreboard and Tomasulo have:**
  - ***In-order issue, out-of-order execution, out-of-order completion***
- **Recall: An interrupt or exception is *precise* if there is a single instruction for which:**
  - **All instructions before that have committed their state**
  - **No following instructions (including the interrupting instruction) have modified any state.**
- **Need way to resynchronize execution with instruction stream (I.e. with issue-order)**
  - **Easiest way is with *in-order completion (i.e. reorder buffer)***
  - **Other Techniques: Future File, History Buffer**

HW support for precise interrupts

- **Concept of Reorder Buffer (ROB):**
  - **Holds instructions in FIFO order, exactly as they were issued**
    - » **Each ROB entry contains PC, dest reg, result, exception status**
  - **When instructions complete, results placed into ROB**
    - » **Supplies operands to other instruction between execution complete & commit ⇒ more registers like RS**
    - » **Tag results with ROB buffer number instead of reservation station**
  - **Instructions commit ⇒values at head of ROB placed in registers**
  - **As a result, easy to undo speculated instructions on mispredicted branches or on exceptions**



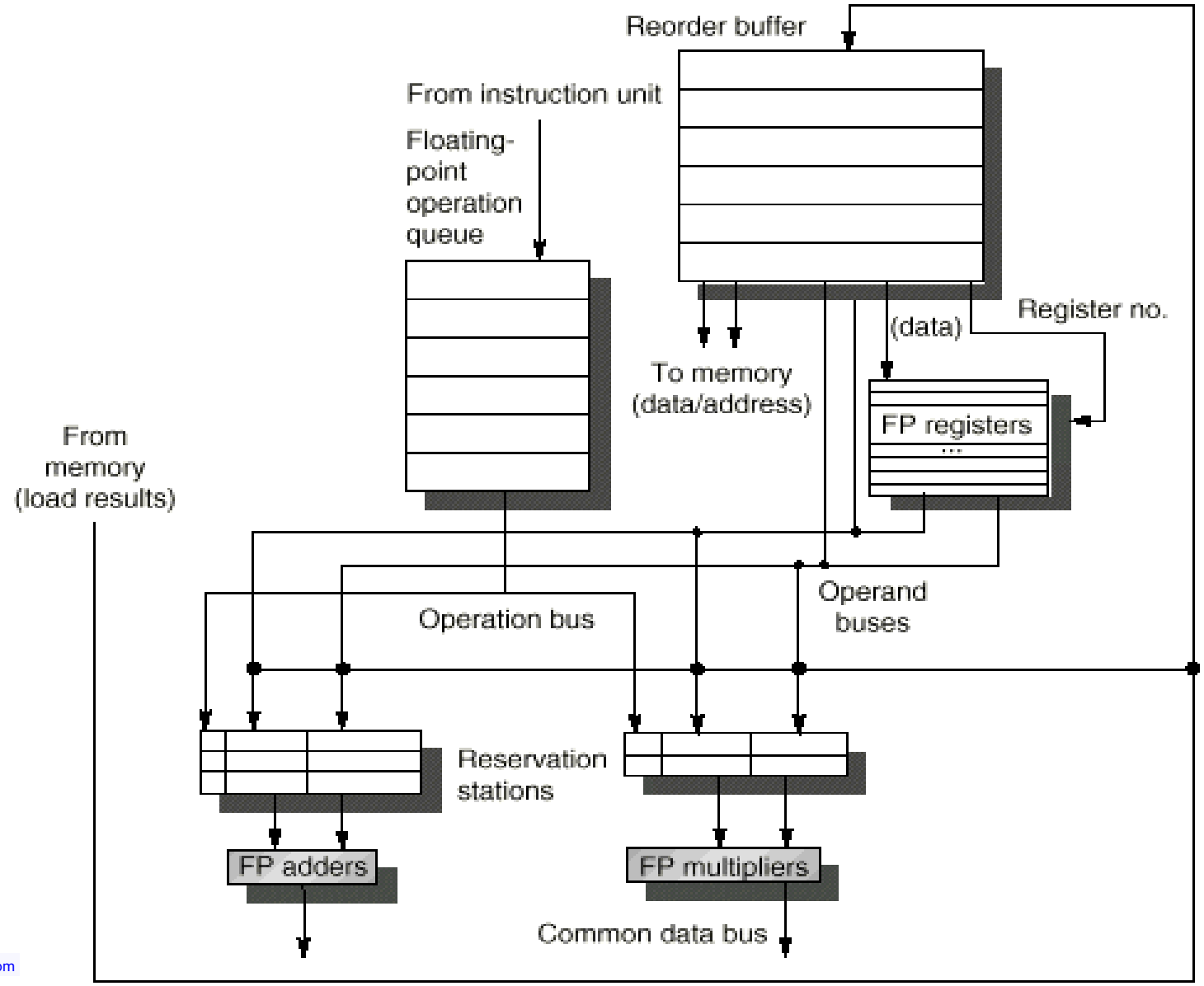
Adding Speculation to Tomasulo

- 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

Tomasulos with Speculation



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**

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")

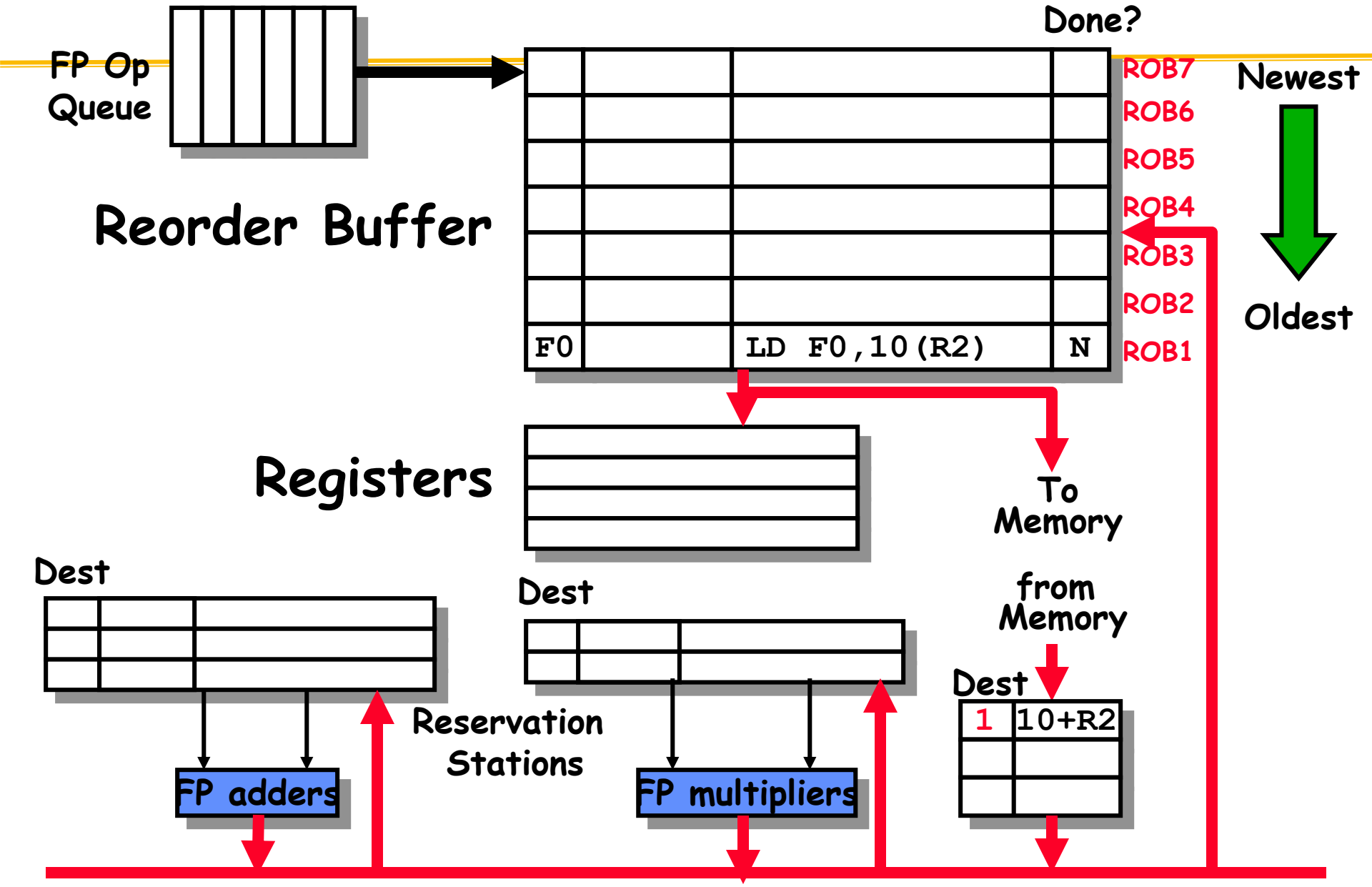
3. Write result—finish execution (WB)

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

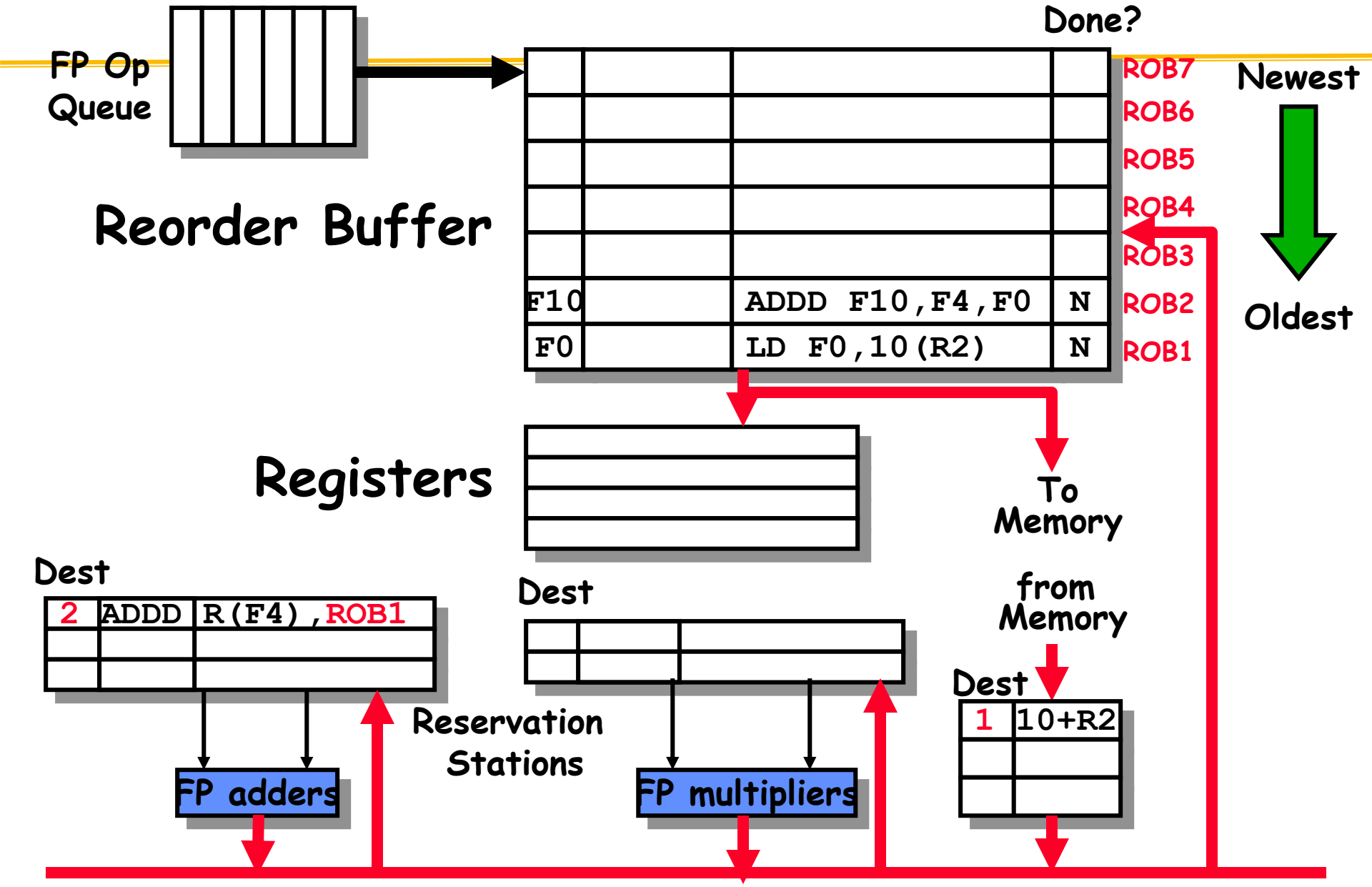
4. Commit—update register with reorder result

   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")

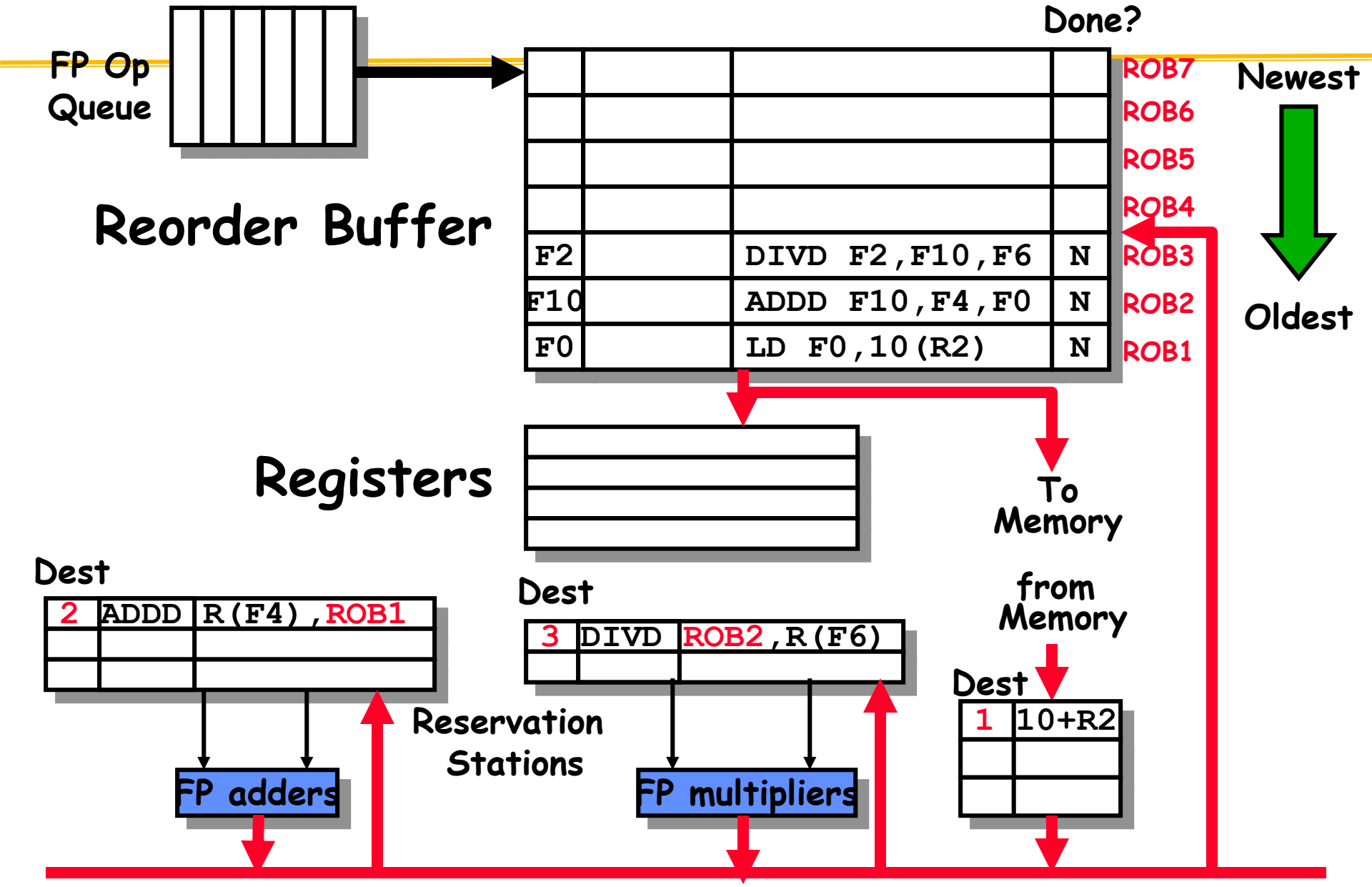
Tomasulo With Reorder buffer:



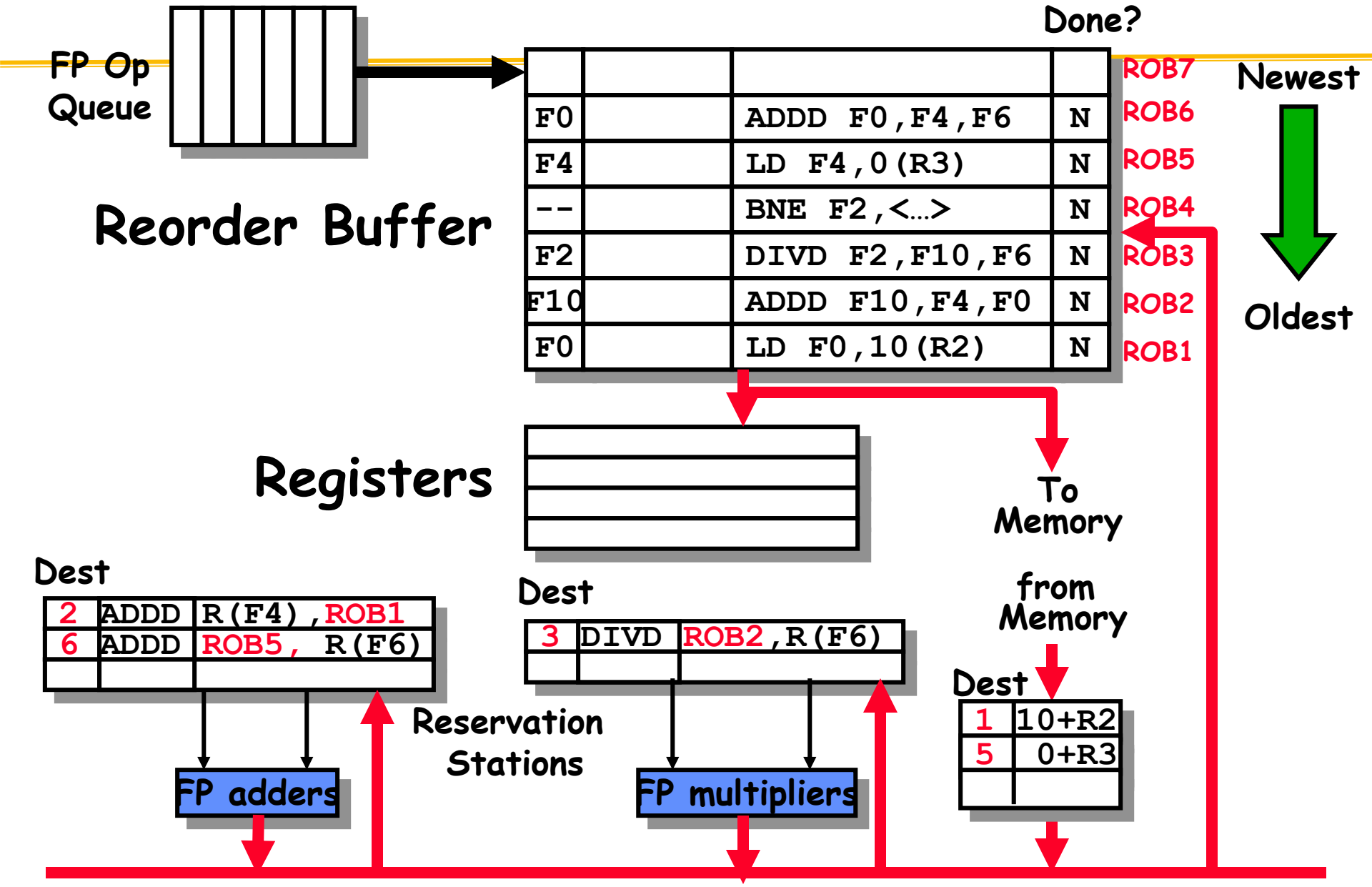
Tomasulo With Reorder buffer:



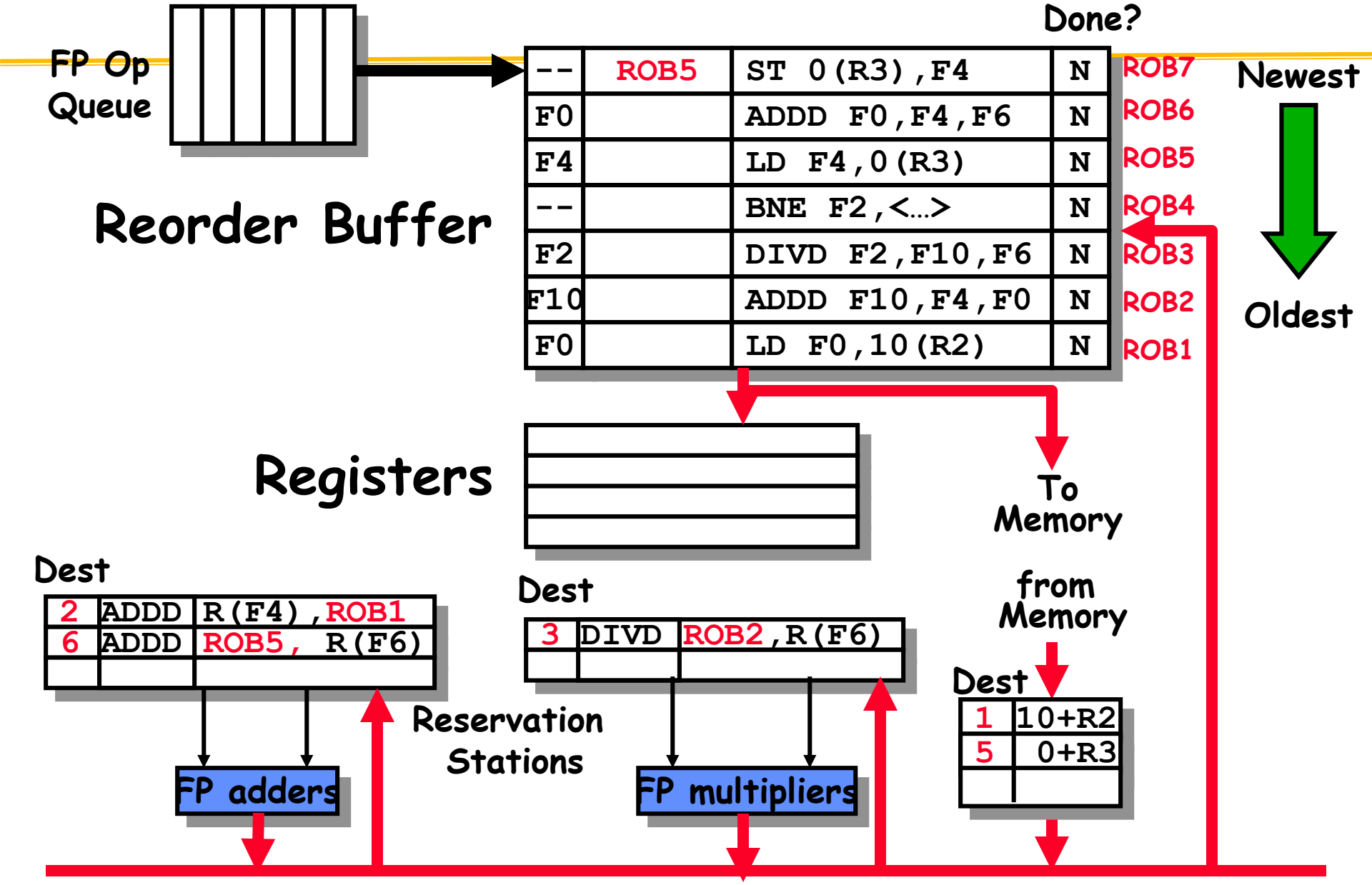
Tomasulo With Reorder buffer:



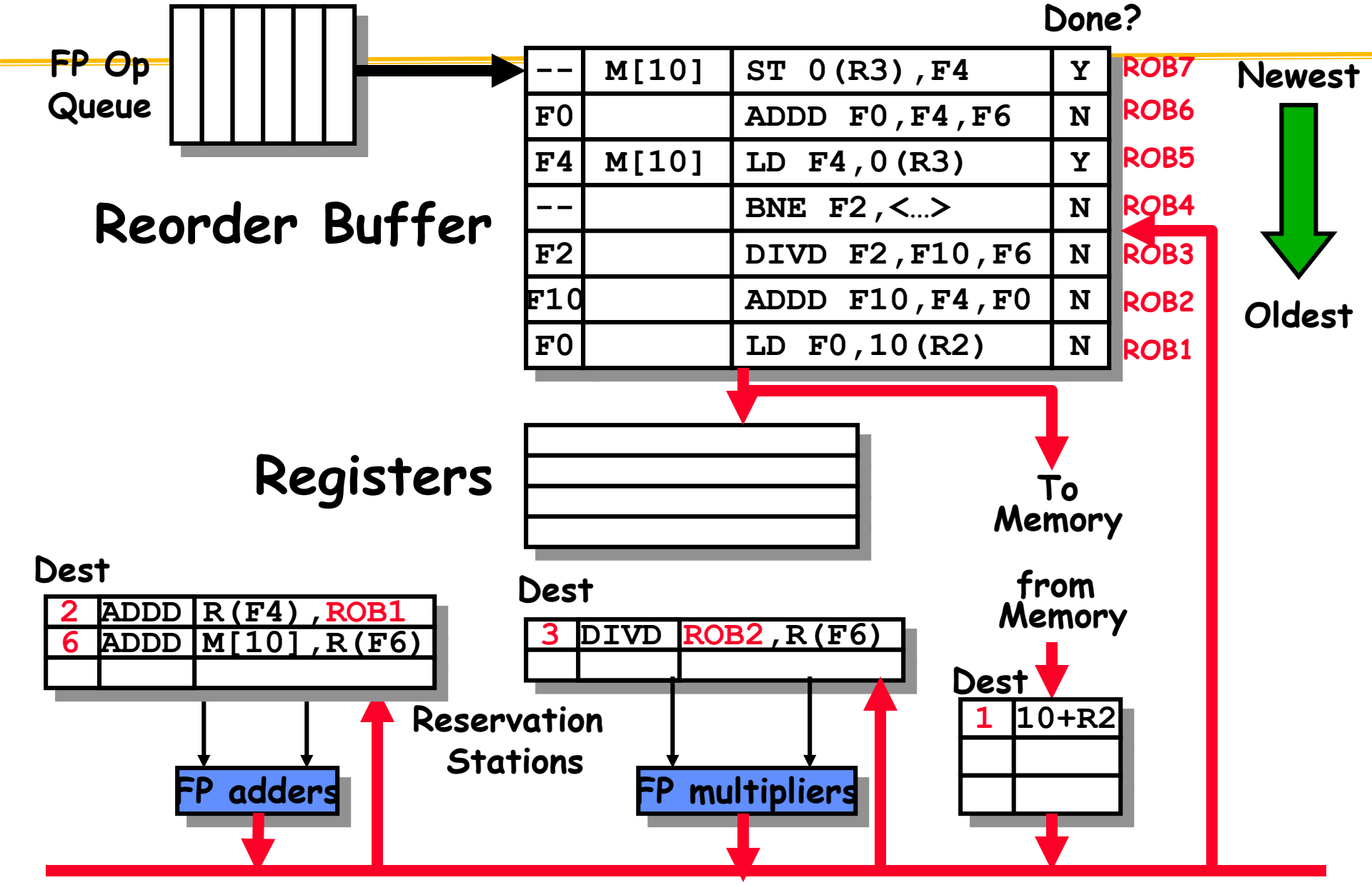
Tomasulo With Reorder buffer:



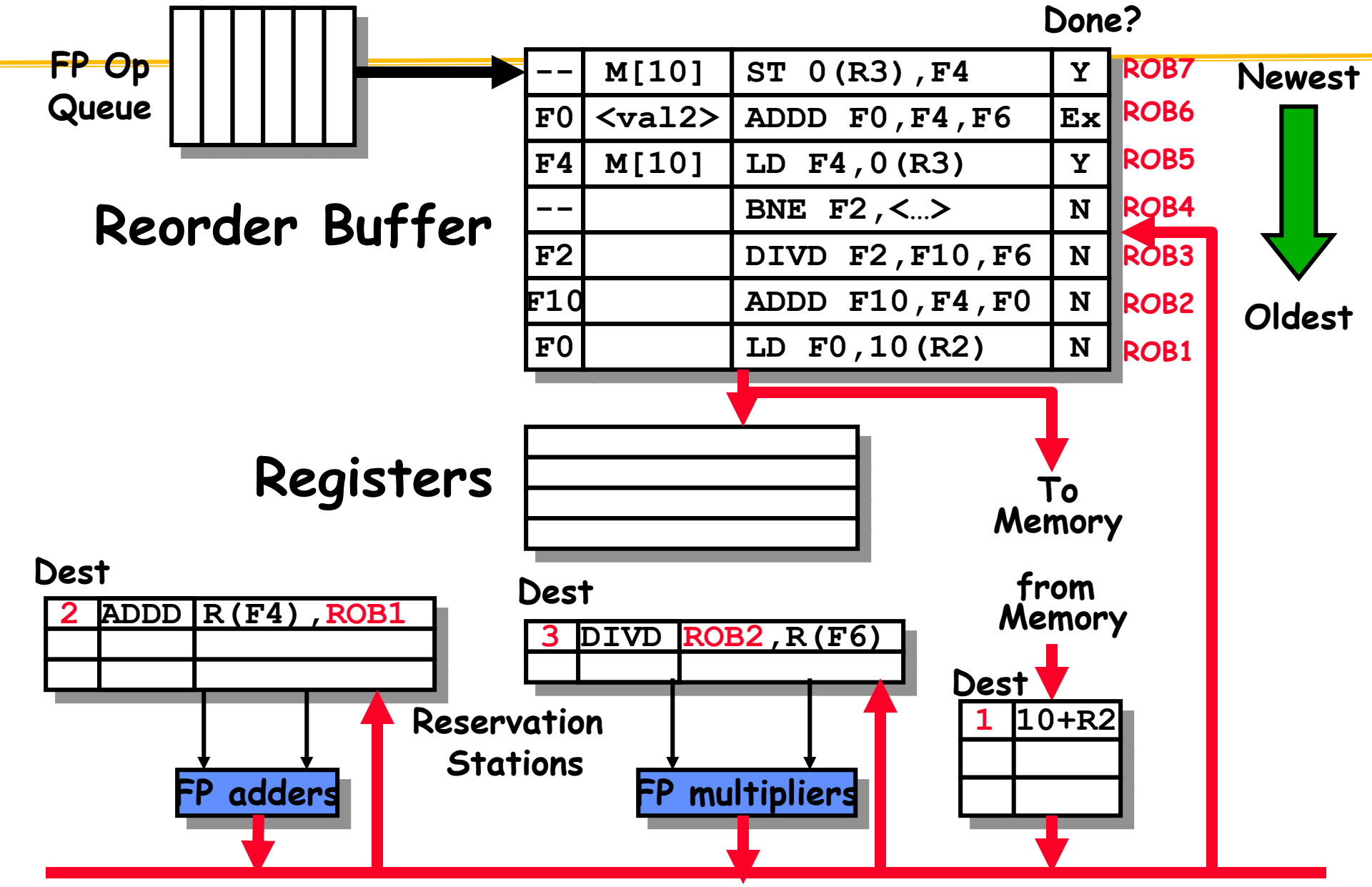
Tomasulo With Reorder buffer:



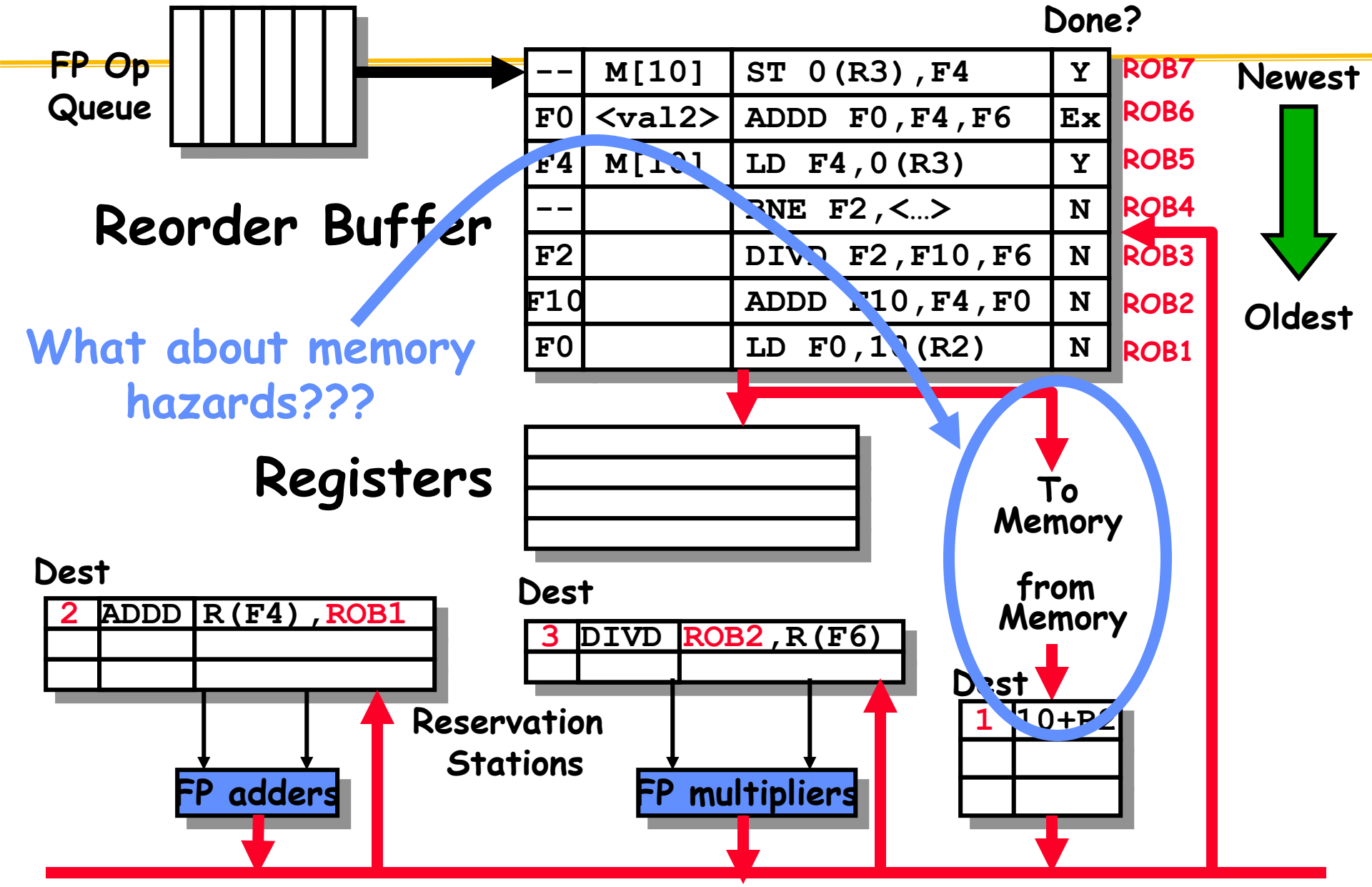
Tomasulo With Reorder buffer:



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**

Memory Disambiguation: Sorting out RAW Hazards in memory

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

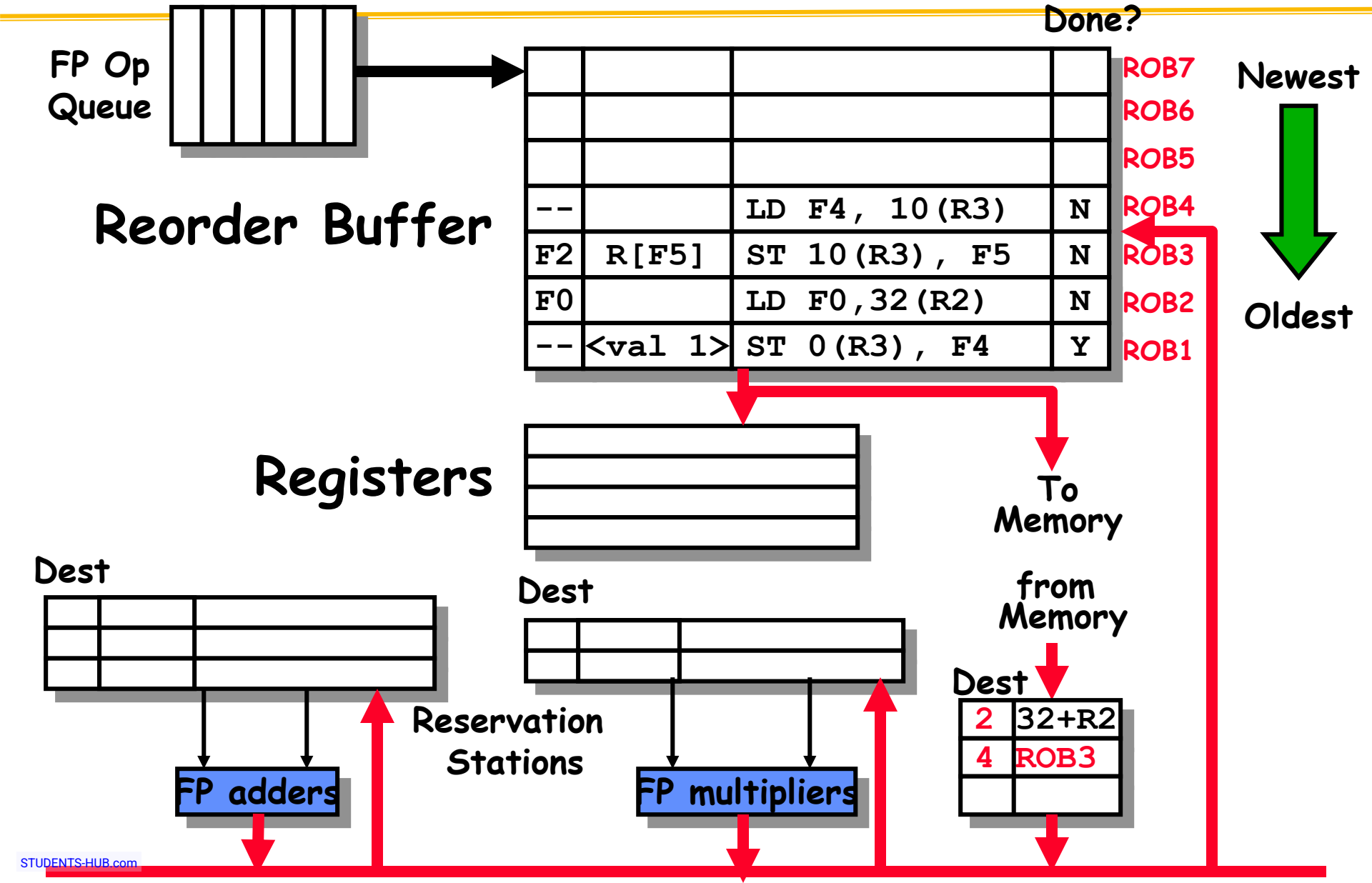
```
Eg:     st      0(R2),R5
        ld      R6,0(R3)
```

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

Hardware Support for Memory Disambiguation

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

Memory Disambiguation:



How much to speculate?

- Speculation Pro: uncover events that would otherwise stall the pipeline (cache misses)
- Speculation Con: speculate costly if exceptional event occurs when speculation was incorrect
- Typical solution: speculation allows only low-cost exceptional events (1st-level cache miss)
- When expensive exceptional event occurs, (2nd-level cache miss or TLB miss) processor waits until the instruction causing event is no longer speculative before handling the event
- Assuming single branch per cycle: future may speculate across multiple branches!

Loop unrolling

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

Instruction producing result	*Instruction using result*	*Latency in cycles*	*stalls between in cycles*
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
```

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

- 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

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

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)
3        ADD.D   F4,F0,F2
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
```

1 cycle stall

2 cycles stall

Rewrite loop to minimize stalls?

27 clock cycles, or 6.75 per iteration

(Assumes R1 is multiple of 4)

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

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 mod 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

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

Branch Prediction

Branch Prediction

- Guess the direction of a branch
- Guess its target if necessary
- Fetch instructions from there
- Execute Speculatively
  - Without knowing whether we should
- Eventually, verify if prediction was correct
  - If correct, good for us
  - if not, well, discard and execute down the right path

For Example

```
while (l)
    if (l->data == 0)
            l->data++;
    l = l->next

loop:   beq r1, r0, done
        ld      r2, 0(r1)
        bne     r2, r0, noinc
inc:    add     r2, r2, 1
        st      r2, 0(r1)
noinc:  ld      r1, 4(r1)
        bra     loop
done:
```

Branch Prediction Steps

- **Elements of Branch Prediction**
  - **Start with branch PC and answer:**
  - **Why just PC? Early in the pipeline!**
  - **Q1? Branch taken or not?**
  - **Q2? Where to?**
  - **Q3? Target Instruction**
- **All must be done to be successful**
- **Let's consider these separately**

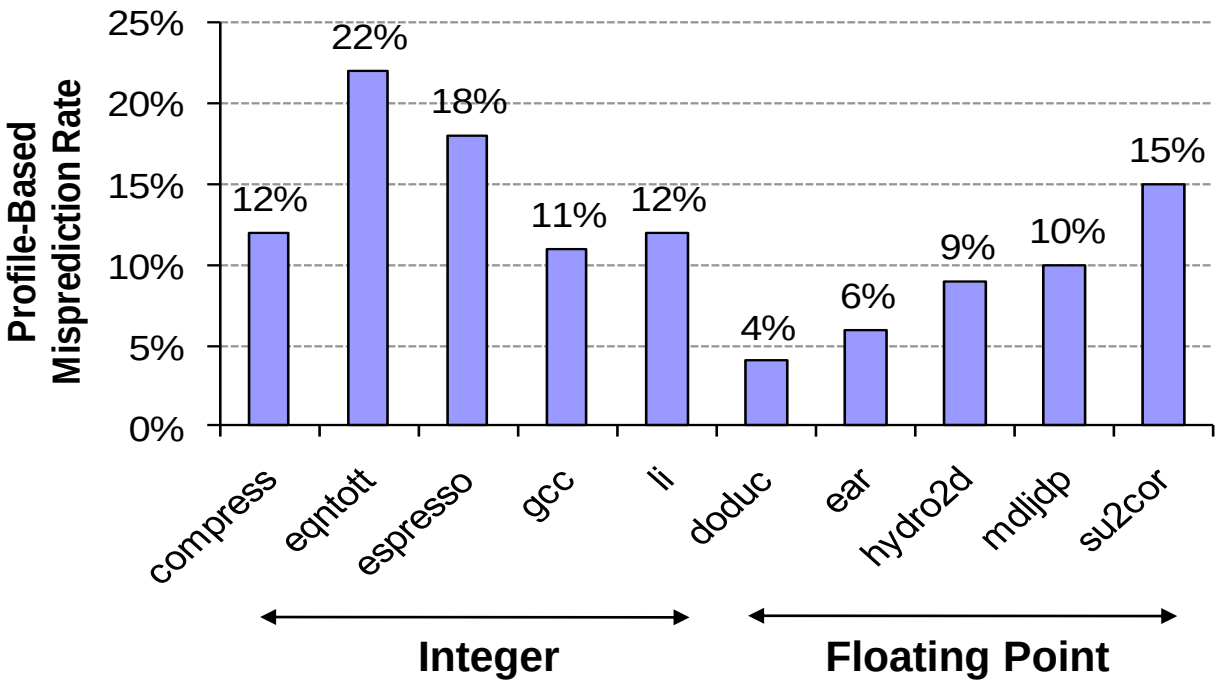
Static Branch Prediction

- **Static:**
  - Decisions do not take into account dynamic behavior
  - Non-adaptive can be another term
- **Always Taken**
- **Always Not-Taken**
- **Forward NT Backward T**
- **If X then T but if Y then NT but if Z then T**
  - More elaborate schemes are possible
- **Bottom line**
  - Accuracy is high but not high enough
  - Say it's 60%

Static Branch Prediction

- **Predict branch statically when we compile the program**
- **Simplest scheme is to predict all branches as taken**
  - **Untaken branch frequency = 34% of all branch instructions (SPEC programs)**
  - **Or predict backward branches as taken and forward branches as not taken**
- **Some processors allow branch prediction hints to be inserted in code**

More accurate static scheme predicts branches using profile information collected from earlier runs, and modify prediction based on last run



Dynamic Branch Prediction

- **Why? Larger window -> More opportunity for parallelism**
- **Basic Idea:**
  - **hardware guesses whether a branch will be taken, and if so where it will go**
- **What makes these work?**
  - **Past Branch Behavior STRONG indicator of future branch behavior**
- **Branches tend to exhibit regular behavior**

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**

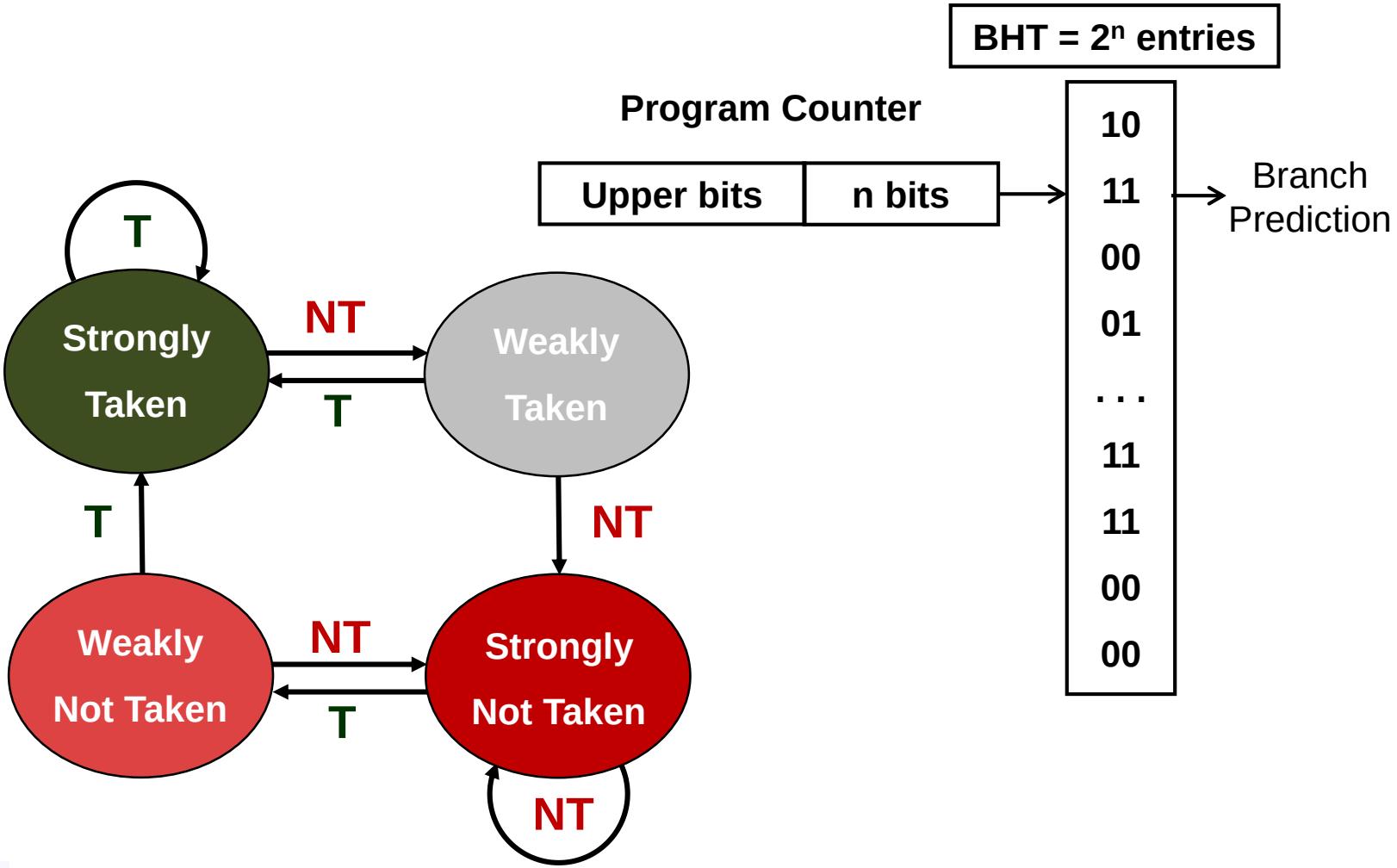
- **Is dynamic branch prediction better than static branch prediction?**
  - **Seems to be**
  - **There are a small number of important branches in programs which have dynamic behavior**

Dynamic Branch Prediction

- Performance = ƒ(accuracy, cost of misprediction)
- Branch History Table (BHT)
- Lower bits of PC address = index to BHT table
  - Each entry consists of few bits
  - Says whether or not branch is predicted to be taken
  - No address check
- 1-bit BHT is simplest to implement
  - Record last branch outcome and uses it to predict future
  - Problem: in a loop, 1-bit BHT will cause two mispredictions
  - 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

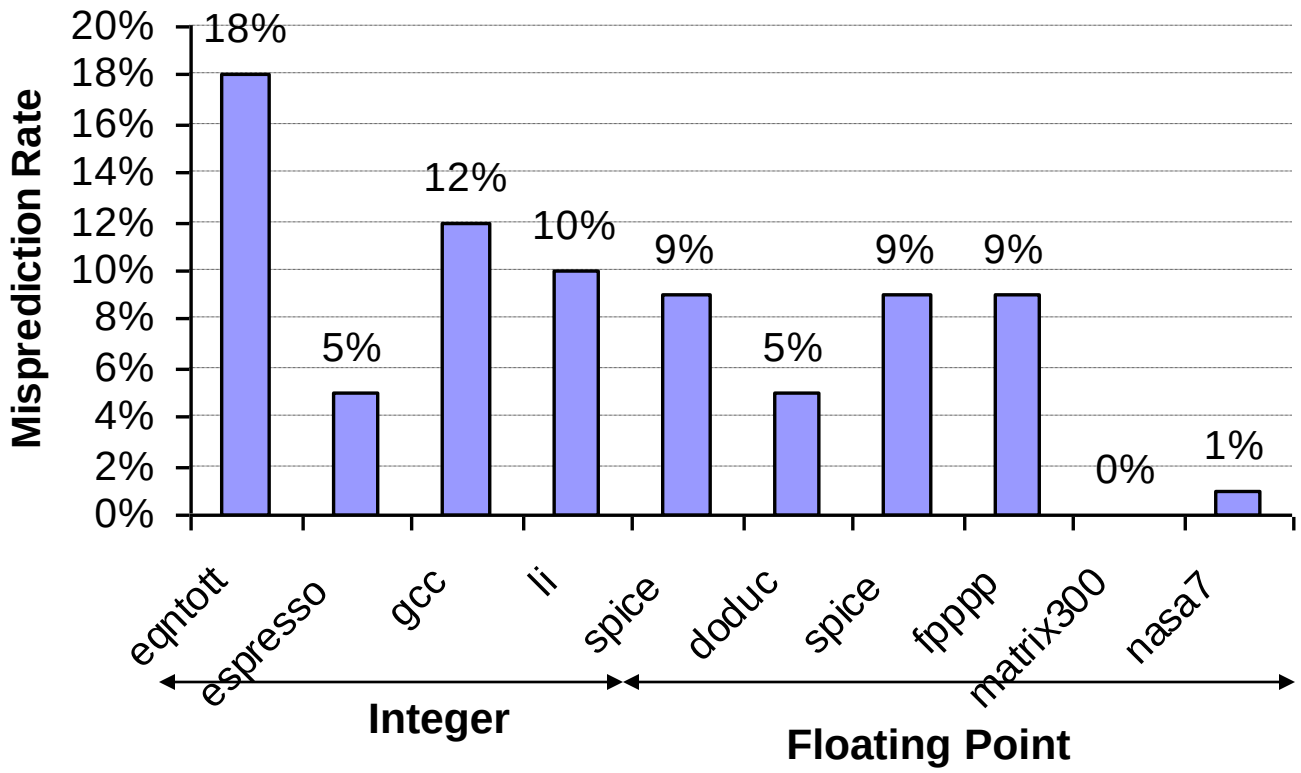
2-bit Predictors

- 2-bit scheme change prediction only if we get two mispredictions



BHT Accuracy

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



Correlating Predictors

- **Branches from different instructions may be correlated**

```
if (aa < 0) aa = 0;
if (bb < 0) bb = 0;
if (aa != bb) { . . . }
```

- **If the first two conditions are true, then the third one will be false**
- **Save history of all recent branch outcomes**
- **Global Branch History Register is a *m*-bit shift register**
  - **Holds most recent *m* branch outcomes**
  - **Approximation to path followed**

Correlating Predictors

- **Here is the MIPS code:**
  - **Assuming that aa and bb are assigned to registers R1 and R2:**

```
        DADDIU    R3,R1,#-2
        BNEZ      R3,L1          jbranch b1     (aa!=2)
        DADD      R1.RO.RO       ;aa=0
L1:     DADDIU    R3,R2,#-2
        BNEZ      R3,L2          ;branch b2     (bb!=Z)
        DADD      R2,R0,R0       ;bb=0
L2:     DSUBU     R3.R1.R2       ;R3=aa-bb
        BEQZ      R3.L3          ;branch b3     (aa==bb)
```

  - **The key observation is that the behavior of branch b3 is correlated with the behavior of branches b1 and b2.**
  - **A predictor that uses only the behavior of a single branch to predict the outcome of that branch can never capture this behavior.**

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

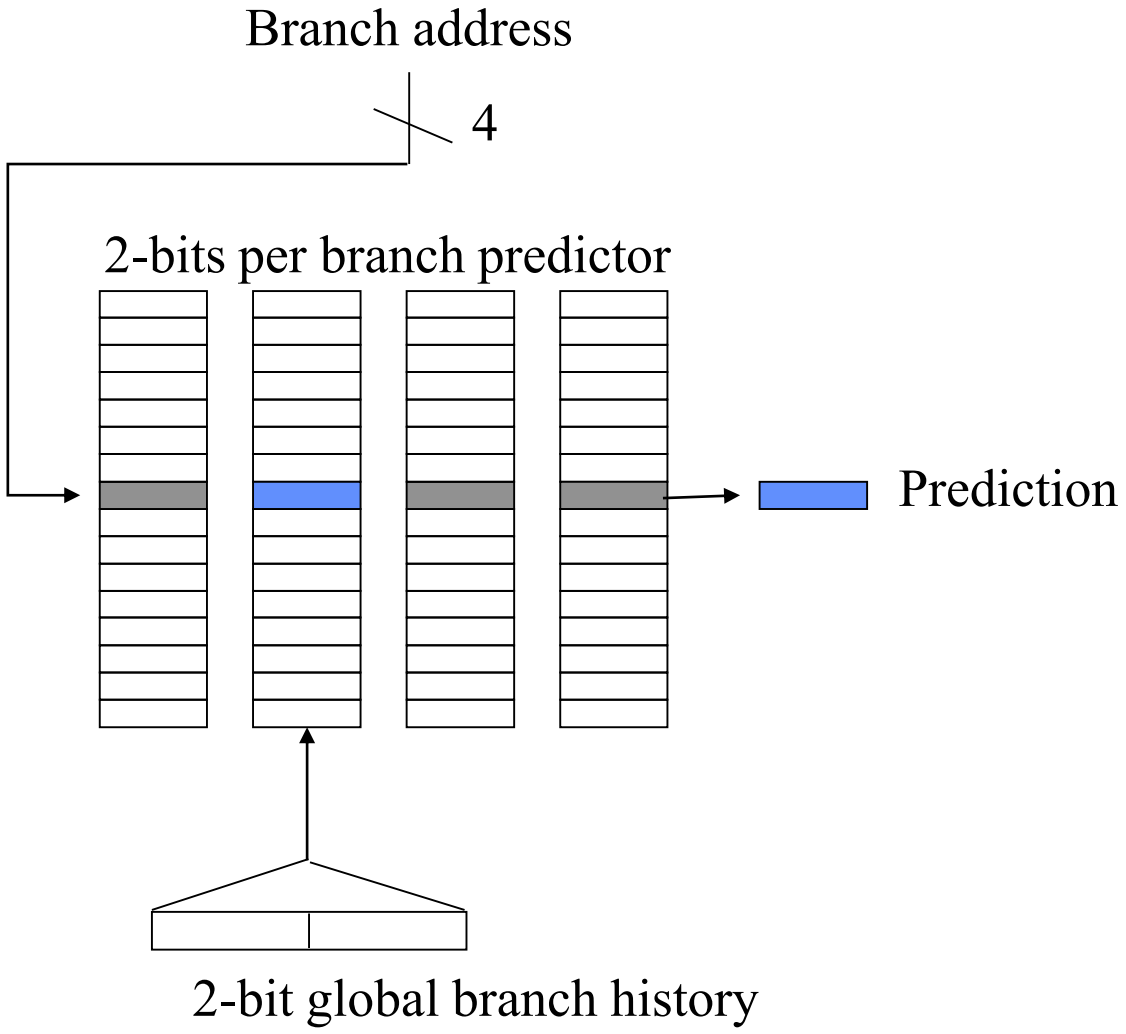
- Global Branch History: *m*-bit shift register keeping T/NT status of last *m* branches.

- Each entry in table has *m* *n*-bit predictors.

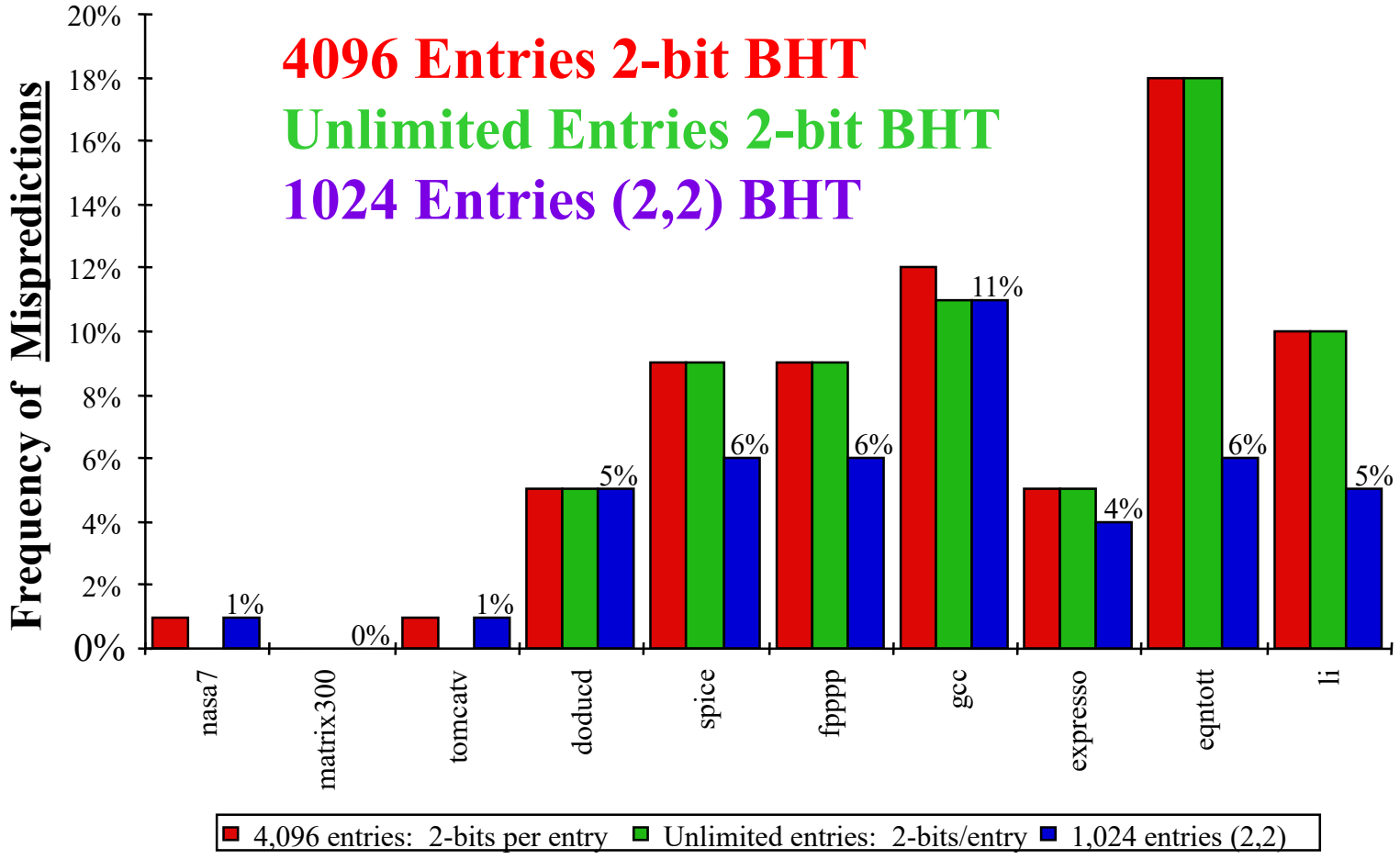
Correlating Branches

(2,2) predictor

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



Accuracy of Different Schemes



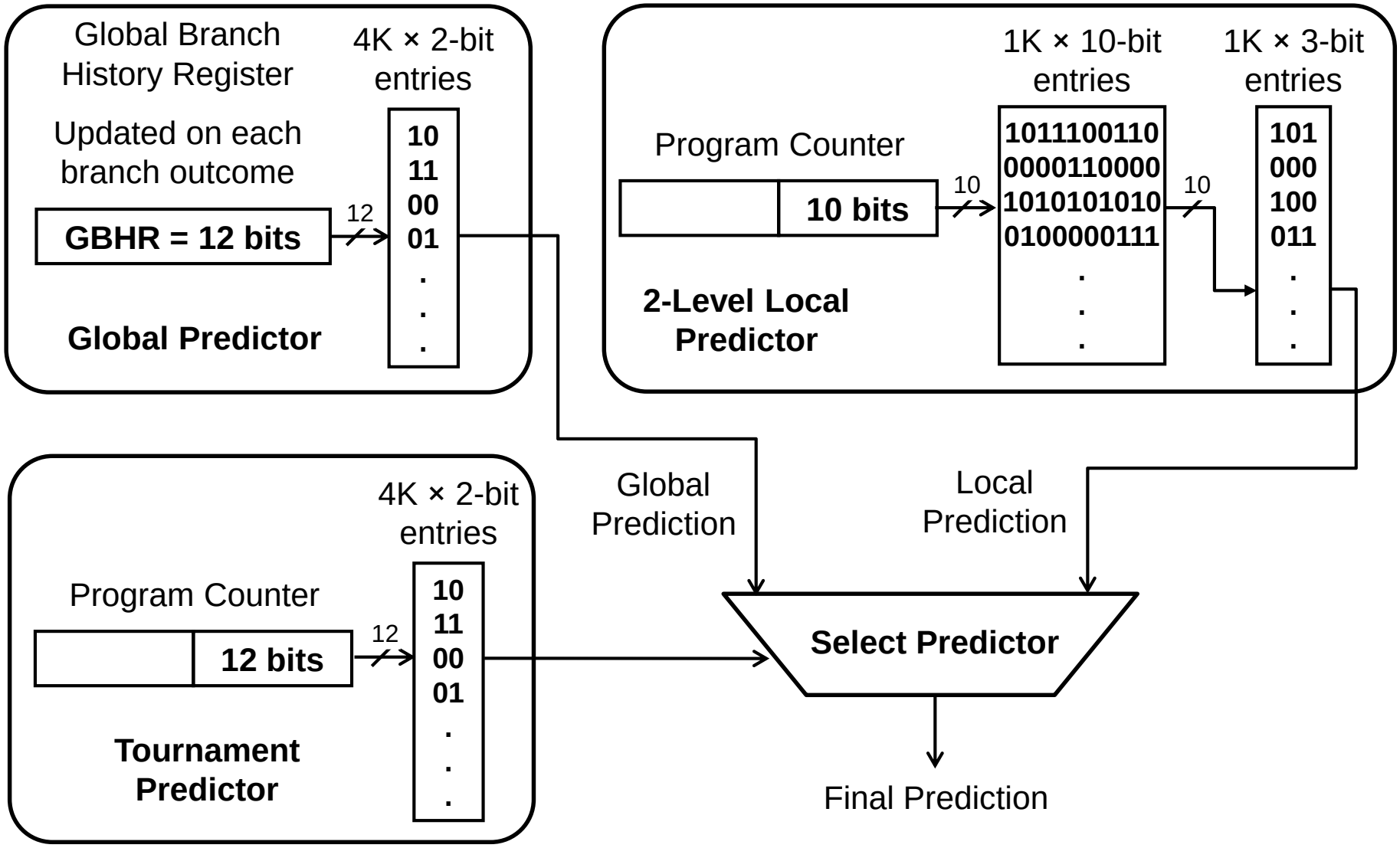
Tournament Predictors

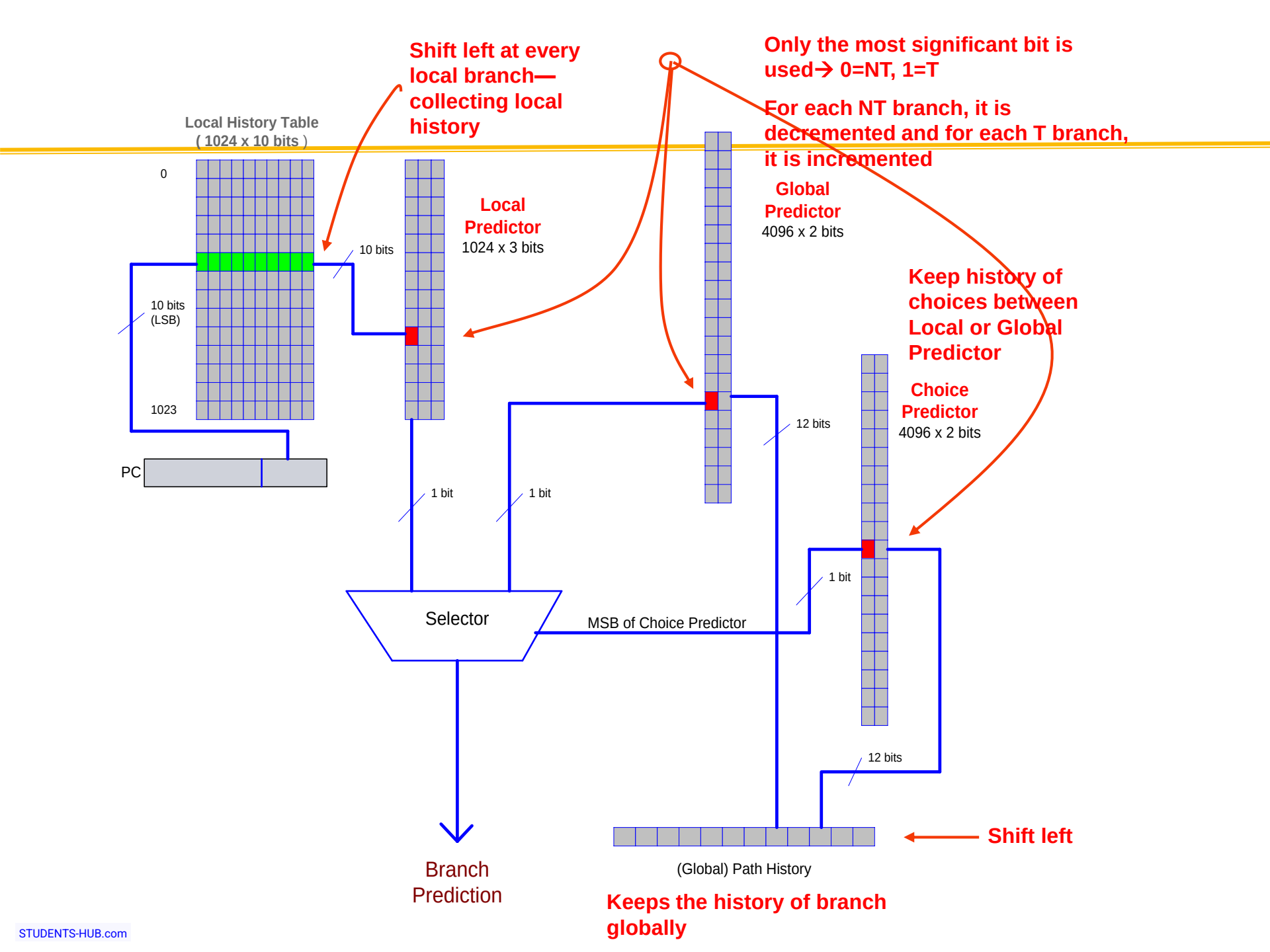
- Multilevel branch predictor
- Use *n*-bit saturating counter to select between predictors
- Usual choice between global and local predictors
- Ability to select the right predictor for a particular branch

Tournament Predictors

- Example of a tournament predictor using 29K bits and used in Alpha 21264, Pentium 4, and Power 5.
- Uses 4K 2-bit counters indexed by local branch address to select between:
- Global predictor
  - 4K entries indexed by history of last 12 branches (2^{12} = 4K)
  - Each entry is a standard 2-bit predictor
- Local predictor is a 2-level predictor
  - Local history table: 1024 10-bit entries recording last 10 branch outcomes, indexed 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

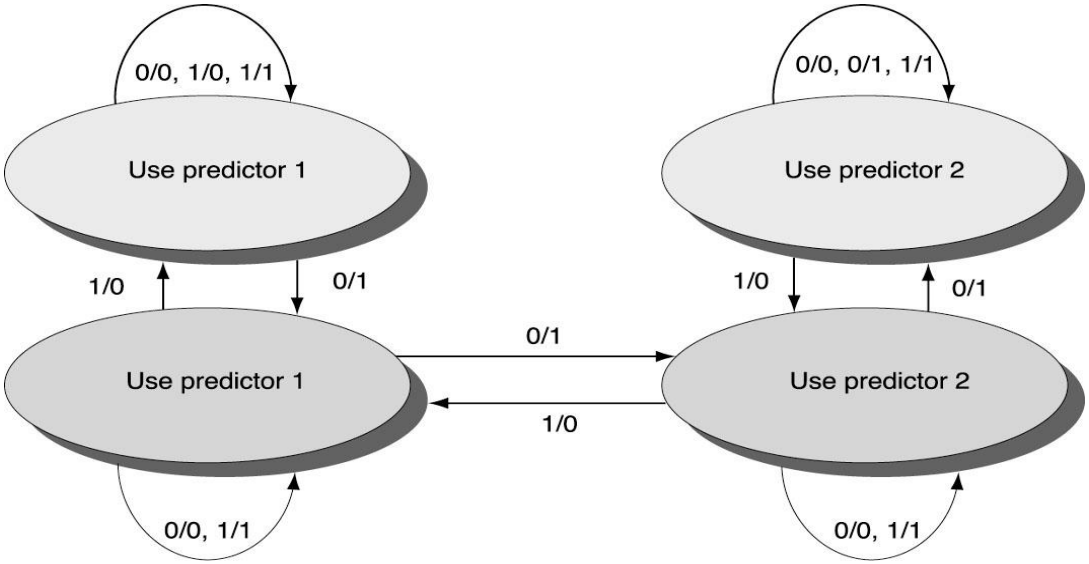
Example of a Tournament Predictor





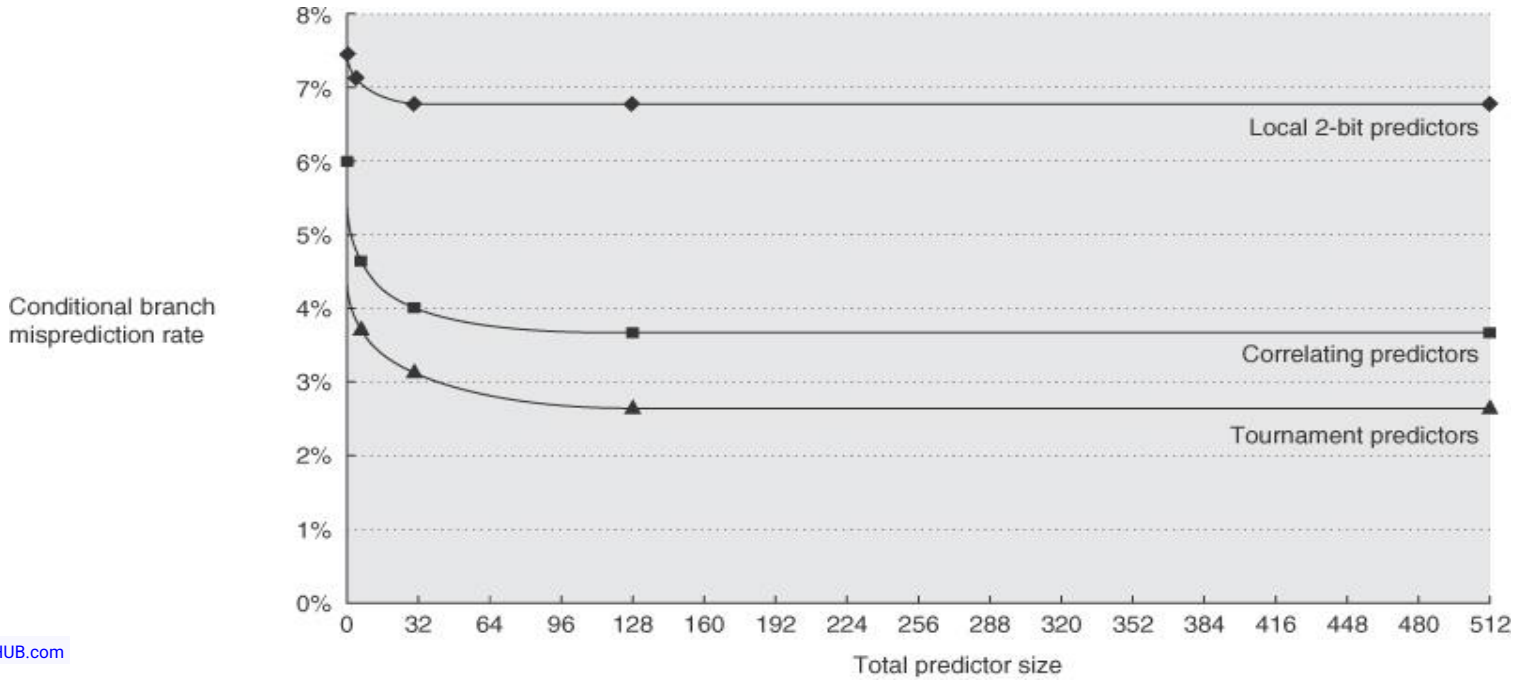
Example of a Tournament Predictor

- The value of the saturating counter determines whether to choose the local or global predictor.
- How does the choice predictor determine this?
  - Whenever the local counter is correct and the global counter is incorrect, the choice predictor's corresponding counter is decremented.
  - Whenever the local counter is incorrect and the global counter is correct, the choice predictor's corresponding counter is incremented.
  - If the counters are both correct, or both incorrect, the choice predictor's corresponding counter is not changed.

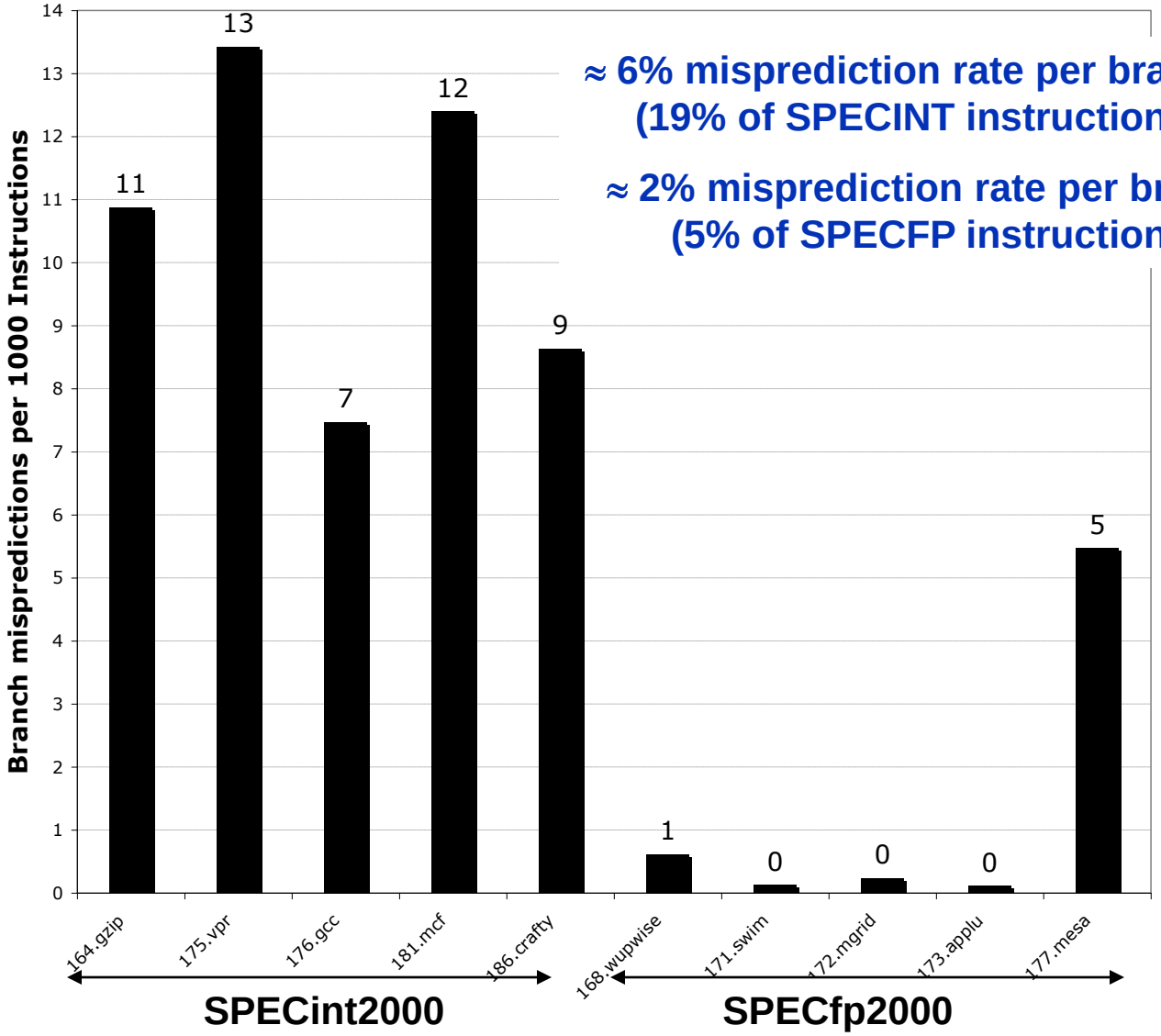


Comparing Predictors

- **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)



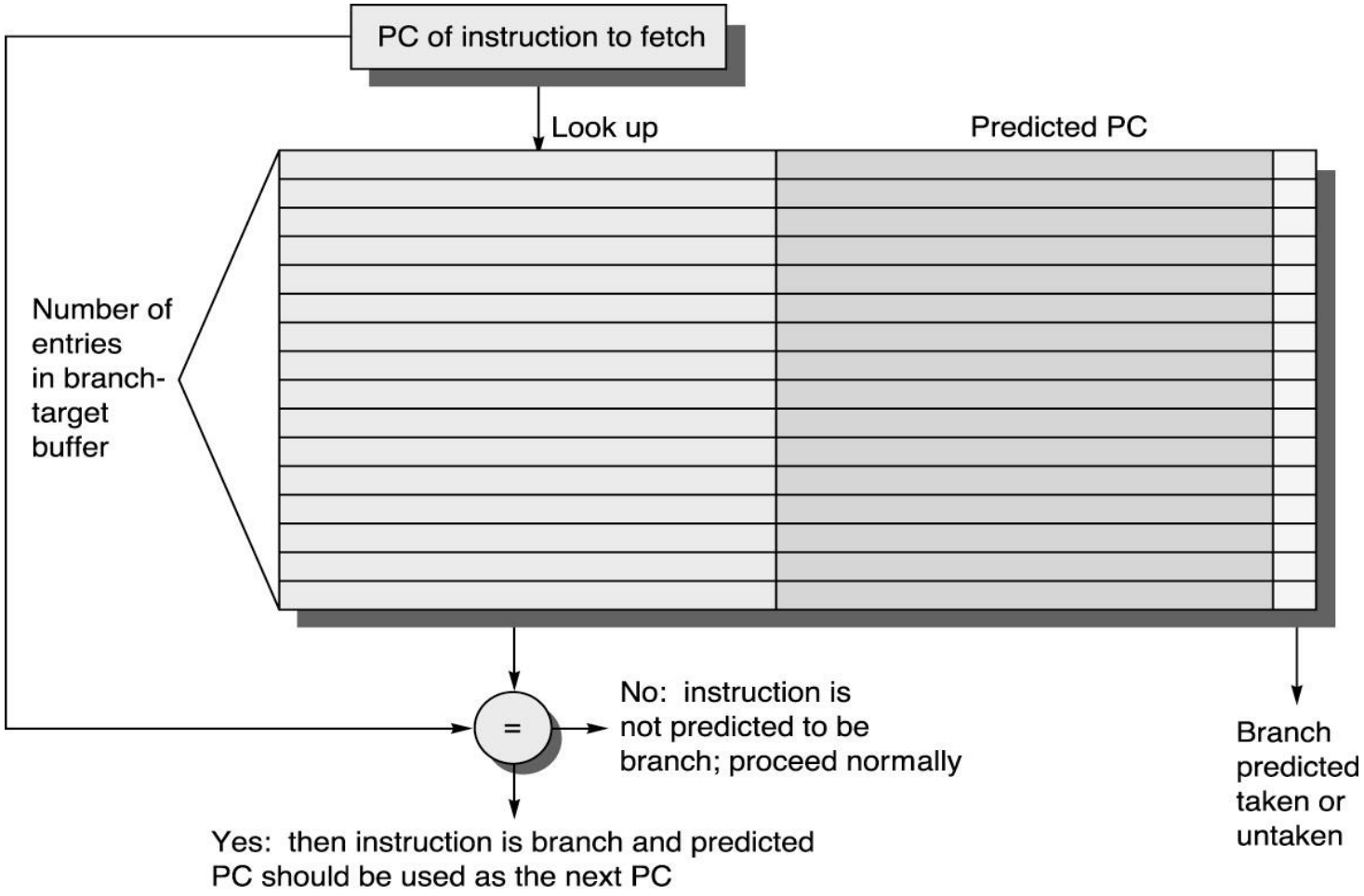
≈ 6% misprediction rate per branch SPECint
(19% of SPECINT instructions are branch)

≈ 2% misprediction rate per branch SPECfp
(5% of SPECFP instructions are branch)

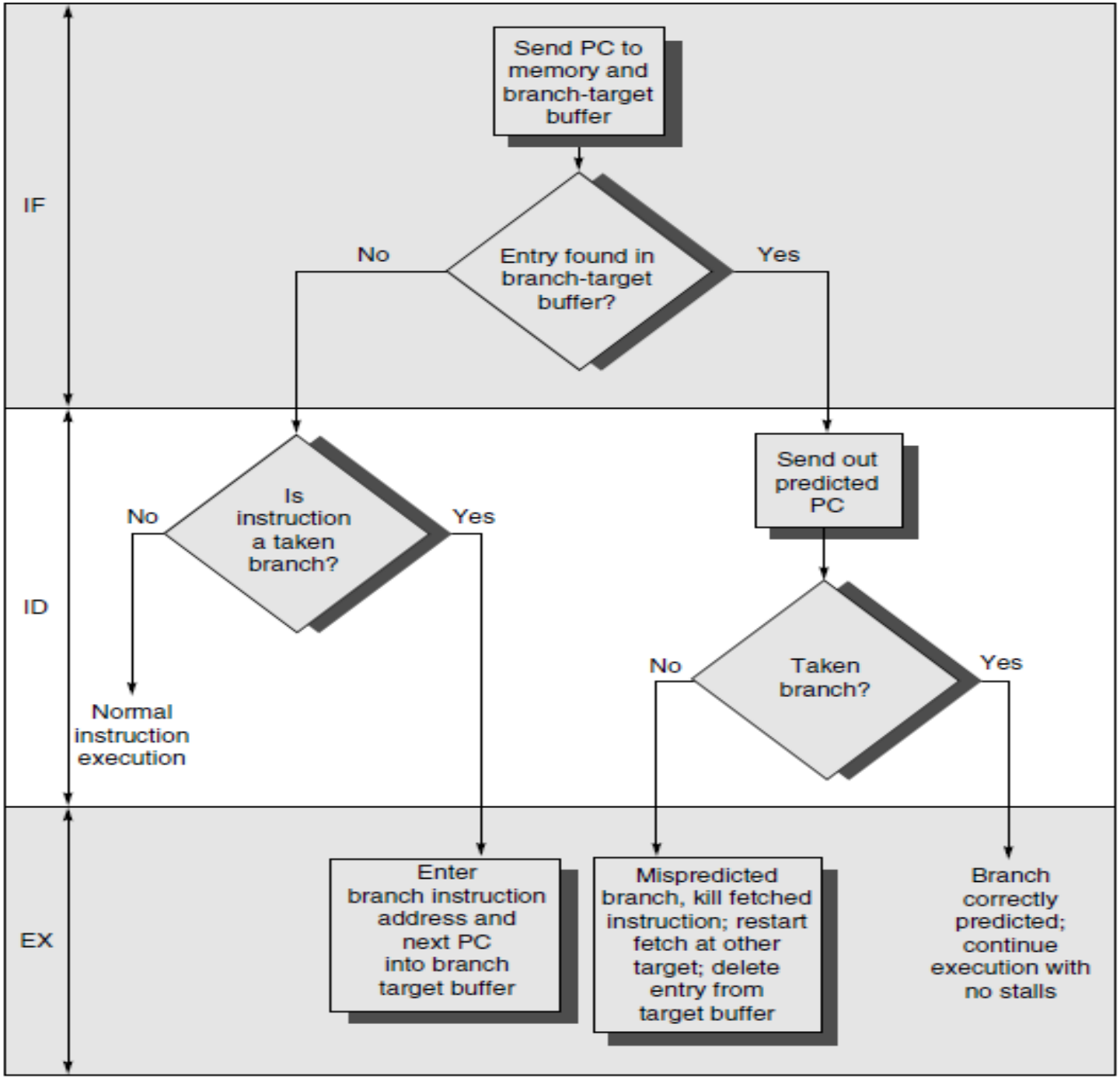
Branch Target Buffers (BTB)

- **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

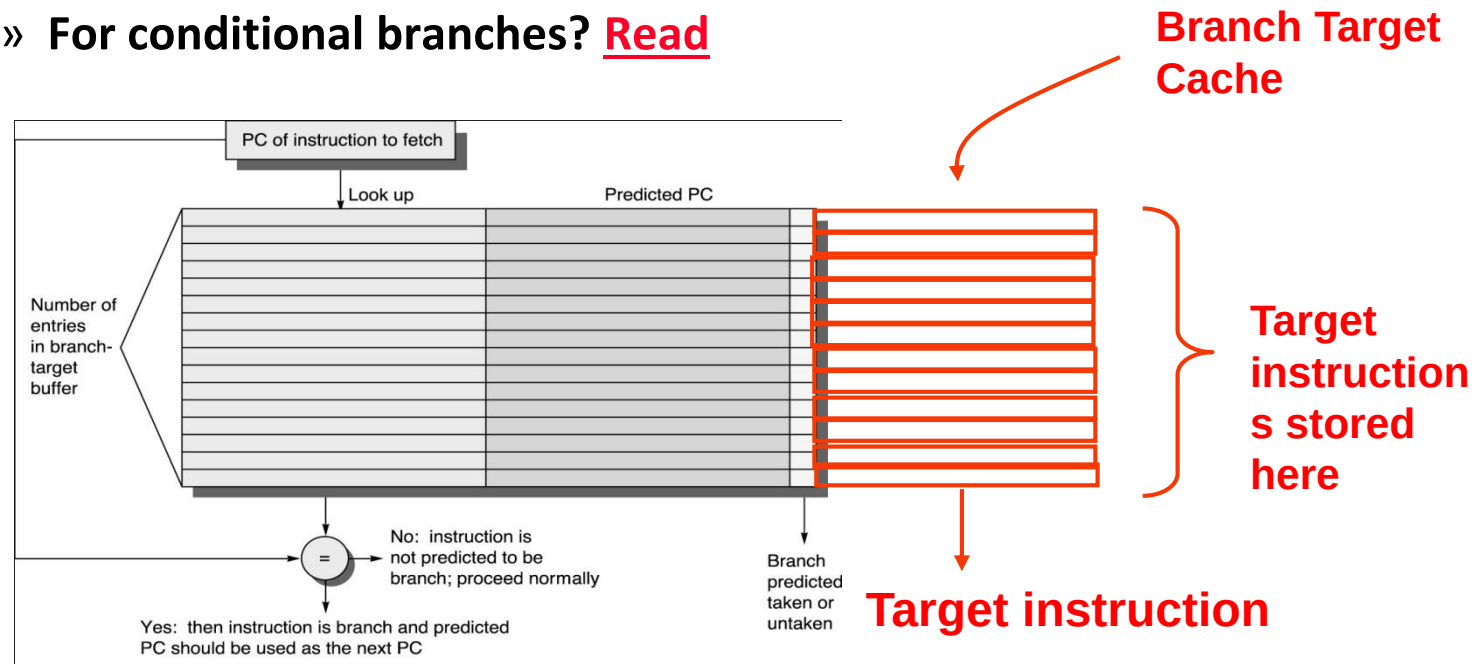


Branch Target Buffers



Branch Target Cache

- **Similar to BTB, but we also want to get the *target instruction*!**
  - **– Prediction returns not just the target address, but also the instruction stored there**
  - **– Allows zero-cycle *unconditional* branches (branch-folding)**
    - **Send target-instruction to ID rather than branch**
    - **Branch is not even sent into pipe**
    - **» For conditional branches? Read**



Return Address Predictors

- Included in many recent processors
  - Alpha 21264 => 12 entry RAS (Return Address Stack)
- Procedure returns account for ~85% of indirect jumps (jumps whose address varies at run time). It will then return to many different locations—BTB may not predict accurately,
- Therefore, small buffer of Return Addresses=cache of the most recent return addresses
- Like a hardware stack, LIFO
  - At Procedure Call => Push Return address 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

Relationship between precise interrupts and speculation:

- Speculation is a form of guessing.
- Important for branch prediction:
  - Need to "take our best shot" at predicting branch direction.
  - If we issue multiple instructions per cycle, lose lots of potential instructions otherwise:
    - » Consider 4 instructions per cycle
    - » If take single cycle to decide on branch, waste from 4 - 7 instruction slots!
- If we speculate and are wrong, need to back up and restart execution to point at which we predicted incorrectly:
  - This is exactly same as precise exceptions!
- Technique for both precise interrupts/exceptions and speculation: *in-order completion or commit*

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**

Getting CPI < 1: Issuing Multiple Instructions/Cycle

- **Vector Processing:** Explicit coding of independent loops as operations on large vectors of numbers
  - Multimedia instructions being added to many processors
- **Superscalar**: varying no. instructions/cycle (1 to 8), scheduled by compiler or by HW (Tomasulo)
  - IBM PowerPC, Sun UltraSparc, DEC Alpha, Pentium III/4
- **(Very) Long Instruction Words (V)LIW**: fixed number of instructions (4-16) scheduled by the compiler; put ops into wide templates
  - Intel Architecture-64 (IA-64) 64-bit address
    - » Renamed: "Explicitly Parallel Instruction Computer (EPIC)"
- Anticipated success of multiple instructions lead to Instructions Per Clock cycle (IPC) vs. CPI

Getting CPI < 1: Issuing Multiple Instructions/Cycle

- **Superscalar MIPS: 2 instructions, 1 FP & 1 anything**
  - **Fetch 64-bits/clock cycle; Int on left, FP on right**
  - **Can only issue 2nd instruction if 1st instruction issues**
  - **More ports for FP registers to do FP load & FP op in a pair**

Type	*Pipe Stages*							
Int. instruction	IF	ID	EX	MEM	WB			
FP instruction		IF	ID	EX	MEM	WB		
Int. instruction		IF	ID	EX	MEM	WB		
FP instruction			IF	ID	EX	MEM	WB	
Int. instruction			IF	ID	EX	MEM	WB	
FP instruction				IF	ID	EX	MEM	WB

- **1 cycle load delay expands to 3 instructions in SS**
  - **instruction in right half can't use it, nor instructions in next slot**

Multiple Issue Issues

- issue packet: group of instructions from fetch unit that could potentially issue in 1 clock
  - If instruction causes structural hazard or a data hazard either due to earlier instruction in execution or to earlier instruction in issue packet, then instruction does not issue
  - 0 to N instruction issues per clock cycle, for N-issue
- Performing issue checks in 1 cycle could limit clock cycle time:
  - => issue stage usually split and pipelined
  - 1st stage decides how many instructions from within this packet can issue, 2nd stage examines hazards among selected instructions and those already been issued
  - => higher branch penalties => prediction accuracy important

Multiple Issue Challenges

- While Integer/FP split is simple for the HW, get CPI of 0.5 only for programs with:
  - Exactly 50% FP operations AND No hazards
- If more instructions issue at same time, greater difficulty of decode and issue:
  - Even 2-scalar => examine 2 opcodes, 6 register specifiers, & decide if 1 or 2 instructions can issue
  - Register file: need 2x reads and 1x writes/cycle
  - Rename logic: must be able to rename same register multiple times in one cycle! For instance, consider 4-way issue:

```
add r1, r2, r3              add p11, p4, p7
sub r4, r1, r2     ⇒        sub p22, p11, p4
lw  r1, 4(r4)               lw  p23, 4(p22)
add r5, r1, r2              add p12, p23, p4
```

    Imagine doing this transformation in a single cycle!
  - Result buses: Need to complete multiple instructions/cycle
    - » So, need multiple buses with associated matching logic at every reservation station.
    - » Or, need multiple forwarding paths

Superscalar Dynamic Scheduling

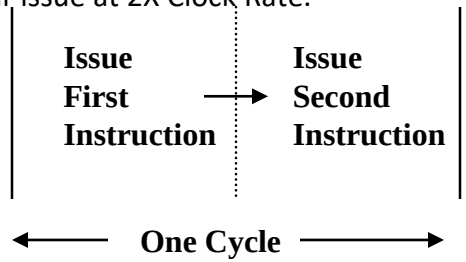
- **The Tomasulo dynamic scheduling algorithm is extended to issue more than one instruction per cycle.**
- **However the restriction that instructions must issue in program order still holds to avoid violating instruction dependencies (construct correct dependency graph dynamically).**
  - The result of issuing multiple instructions in one cycle should be the same as if they were single-issued, one instruction per cycle.
- **How to issue two instructions and keep in-order instruction issue for Tomasulo?**
- **Simplest Method: Restrict Type of Instructions Issued Per Cycle**
- **To simplify the issue logic, issue one one integer + one floating-point instruction per cycle (for a 2-way superscalar).**
  - 1 Tomasulo control for integer, 1 for floating point.
- **FP loads/stores might cause a dependency between integer and FP issue:**
  - Replace load reservation stations with a load queue; operands must be read in the order they are fetched (program order).
  - Replace store reservation stations with a store queue; operands must be written in the order they are fetched.
    - » **Load checks addresses in Store Queue to avoid RAW violation**
      - **(get load value from store queue if memory address matches)**
    - » **Store checks addresses in Load Queue to avoid WAR, and checks Store Queue to avoid WAW.**

Superscalar Dynamic Scheduling

Three techniques can be used to support multiple instruction issue in Tomasulo without putting restrictions on the type of instructions issued per cycle:

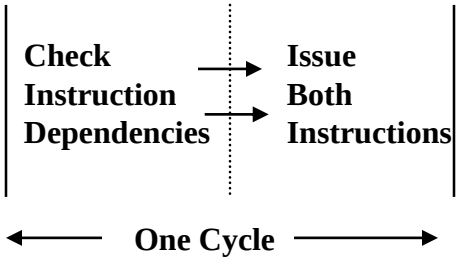
1 **Issue at a higher clock rate so that issue remains in order.**

- For example for a 2-Issue supercalar issue at 2X Clock Rate.



2 **Widen the issue logic to handle multiple instruction issue**

- All possible dependencies between instructions to be issues are detected at once and the result of the multiple issue matches in-order issue



2-Issue superscalar

0, 1 or 2 instructions issued per cycle for either method

Why?

For correct dynamic construction of dependency graph:
The result of issuing multiple instructions in one cycle should be the same as if they were single-issued, one instruction per cycle.

Superscalar Dynamic Scheduling

3 **To avoid increasing the CPU clock cycle time in the last two approaches, multiple instruction issue can be spilt into two pipelined issue stages:**

- Issue Stage One: Decide how many instructions can issue simultaneously checking dependencies within the group of instructions to be issued + available RSs, ignoring instructions already issued.
- Issue Stage Two: Examine dependencies among the selected instructions from the group and the those already issued.

- **This approach is usually used in dynamically-scheduled wide superscalars that can issue four or more instructions per cycle.**
- **Splitting the issue into two pipelined staged increases the CPU pipeline depth and increases branch penalties**
  - This increases the importance of accurate dynamic branch prediction methods.
- **Further pipelining of issue stages beyond two stages may be necessary as CPU clock rates are increased.**
- **The dynamic scheduling/issue control logic for superscalars is generally very complex *growing at least quadratically* with issue width.**
  - e.g 4 wide superscalar -> 4x4 = 16 times complexity of single issue CPU

Superscalar Dynamic Scheduling with Dual-issue

Iteratic n numbe r	Instructions		Issues at clock cycle number	Executes at clock cycle number	Memory access at clock cycle number	Write CDB *i* t clock cycle number	Comment
1	LD	R2,0(R1)	1	2	3	4	First issue
1	DADDIU	R2,R2,#1	1	5		6	Wait for LW
1	SD	R2,0(R1)	2	3	7		Wait for DADDIU
1	DADDIU	R1,R1,#8	2	3		4	Execute directly
1	BNE	R2,R3,L00P	3	7			Wait for DADDIU
2	LD	R2,0(R1)	4	8	9	10	Wait for BNE
2	DADDIU	R2,R2,#1	4	11		12	Wait for LW
2	SD	R2,0(R1)	5	9	13		Wait for DADDIU
2	DADDIU	R1,R1,#8	5	8		9	Wait for BNE
2	BNE	R2,R3,L00P	6	13			Wait for DADDIU
3	LD	R2,0(R1)	7	14	15	16	Wait for BNE
3	DADDIU	R2,R2,#1	7	17		18	Wait for LW
3	SD	R2,0(R1)	8	15	19		Wait for DADDIU
3	DADDIU	R1,R1,#8	8	14		15	Wait for BNE
3	BNE	R2,R3,L00P	9	19			Wait for DADDIU

Superscalar Dynamic Scheduling with Dual-issue and speculation

Iter. #	Instructions	Issues at clock #	Executes at clock #	Read access at clock #	Write CDB at clock #	Com-mits at clock #	Comment
1	LW R2,0(R1)	1	2	3	4	5	First issue
1	DADDIU R2,R2,#1	1	5		6	7	Wait for LW
1	SW 0(R1),R2	2	3			7	Wait for DADDIU
1	DADDIU R1,R1,#4	2	3		4	8	Commit in order
1	BNE R2,R3,LOOP	3	7			8	Wait for ADDDI
2	LW R2,0(R1)	4	5	6	7	9	No execute delay
2	DADDIU R2,R2,#1	4	8		9	10	Wait for LW
2	SW 0(R1),R2	5	6			10	Wait for DADDIU
2	DADDIU R1,R1,#4	5	6		7	11	Commit in order
2	BNE R2,R3,LOOP	6	10			11	Wait for DADDIU
3	LW R2,0(R1)	7	8	9	10	12	Earliest possible
3	DADDIU R2,R2,#1	7	11		12	13	Wait for LW
3	SW 0(R1),R2	8	9			13	Wait for DADDIU
3	DADDIU R1,R1,#4	8	9		10	14	Executes earlier
3	BNE R2,R3,LOOP	9	11			14	Wait for DADDIU

Limits to ILP

- Conflicting studies of amount
  - Benchmarks (vectorized Fortran FP vs. integer C programs)
  - Hardware sophistication
  - Compiler sophistication
- How much ILP is available using existing mechanisms with increasing HW budgets?
- Do we need to invent new HW/SW mechanisms to keep on processor performance curve?
  - Intel MMX, SSE (Streaming SIMD Extensions): 64 bit ints
  - Intel SSE2: 128 bit, including 2 64-bit Fl. Pt. per clock
  - Motorola AltaVec: 128 bit ints and FPs
  - Supersparc Multimedia ops, etc.

Limits to ILP

Initial HW Model here; MIPS compilers.

Assumptions for ideal/perfect machine to start:

1. ***Register renaming*** **– infinite virtual registers**
   => all register WAW & WAR hazards are avoided
2. ***Branch prediction*** **– perfect; no mispredictions**
3. ***Jump prediction*** **– all jumps perfectly predicted**
   2 & 3 => machine with perfect speculation & an unbounded buffer of instructions available
4. ***Memory-address alias analysis*** **– addresses are known & a store can be moved before a load provided addresses not equal**

Also:

unlimited number of instructions issued/clock cycle; perfect caches;
1 cycle latency for all instructions (FP *,/);