

332

**Advanced Computer Architecture
Chapter 3**

**Instruction Level Parallelism and
Dynamic Execution**

January 2010

Paul H J Kelly

These lecture notes are partly based on the course text, Hennessy and Patterson's *Computer Architecture, a quantitative approach* (3rd 4th eds), and on the lecture slides of David Patterson and John Kubiawicz's Berkeley course (CS252)

Data Dependence and Hazards

- ▶ Instr_j is **data dependent** on Instr_i
Instr_j tries to read operand before Instr_i writes it

I: add **r1**,r2,r3
J: sub r4,**r1**,r3



- ▶ or Instr_j is data dependent on Instr_k which is dependent on Instr_i
- ▶ Caused by a “**True Dependence**” (compiler term)
- ▶ If true dependence caused a hazard in the pipeline, called a **Read After Write (RAW) hazard**

Data Dependence and Hazards

- Dependences are a property of **programs**
- Presence of dependence indicates **potential** for a hazard, but actual hazard and length of any stall is a 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
- Today looking at HW schemes to avoid hazard

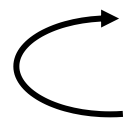
Name Dependence #1: Anti-dependence

▶ **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

▶ There are two kinds:

▶ Name dependence #1: anti-dependence/WAR

▶ Instr_j writes operand before Instr_i reads it

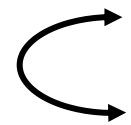
 I: sub r4,**r1**,r3
J: add **r1**,r2,r3
K: mul r6,r1,r7

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**

ILP and Data Hazards

- ▶ HW/SW must preserve **program order**:
order instructions would execute in if executed sequentially 1 at a time as determined by original source program
- ▶ HW/SW goal: exploit parallelism by preserving program order **only where it affects the outcome of the program**
- ▶ 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, two properties critical to program correctness are **exception behavior** and **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:

▶ Problem with moving LW before BEQZ?

▶ **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

Advantages of Dynamic Scheduling

- ▶ **Handles cases when dependences unknown at compile time**
 - ▶ (e.g., because they may involve a memory reference)
- ▶ **It simplifies the compiler**
- ▶ **Allows code that compiled for one pipeline to run efficiently on a different pipeline**
- ▶ **Hardware speculation, a technique with significant performance advantages, that 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**
- We will distinguish when an instruction is **issued**, ***begins execution*** and when it ***completes execution***; between these two times, the instruction is ***in execution***
- In a dynamically scheduled pipeline, all instructions pass through issue stage in order (**in-order issue**)

Dynamic Scheduling Step 1

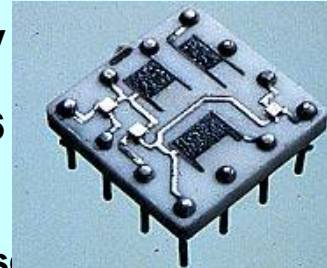
- ▶ Simple pipeline had 1 stage to check both structural and data hazards: Instruction Decode (ID), also called Instruction Issue
- ▶ Split the ID pipe stage of simple 5-stage pipeline into 2 stages:
- ▶ **Issue**—Decode instructions, check for structural hazards
- ▶ **Read operands**—Wait until no data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

A Dynamic Algorithm: Tomasulo's Algorithm

- ▶ For IBM 360/91 (before caches!)
- ▶ 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 a 1966 Computer?
- ▶ The descendants of this have flourished!
 - ▶ Alpha 21264, HP 8000, MIPS 10000/R12000, Pentium II/III/4, Core, Core2, Nehalem, AMD K5,K6,Athlon, Opteron, Phenom, PowerPC 603/604/G3/G4/G5, ...

IBM360/91

- CPU cycle time: 60 nanoseconds
- memory cycle time (to fetch and store eight bytes in parallel): 780ns
- Standard memory capacity: 2,097,152B interleaved 16 ways (magnetic cores)
- Up to 6,291,496 bytes of main storage
- Up to 16.6-million additions/second
- Ca.120K gates, ECL
- Solid Logic Technology (SLT), an IBM invention which encapsulated 5-6 transistors into a small module--a transition technology between discrete transistors and the IC
- About 12 were made



NASA Center for Computational Sciences

See:

*Some Reflections on Computer Engineering:
30 Years after the IBM System 360
Model 91*

Michael J. Flynn

<ftp://arith.stanford.edu/tr/micro30.ps.Z>

Source:

<http://www.columbia.edu/acis/history/36091.html>

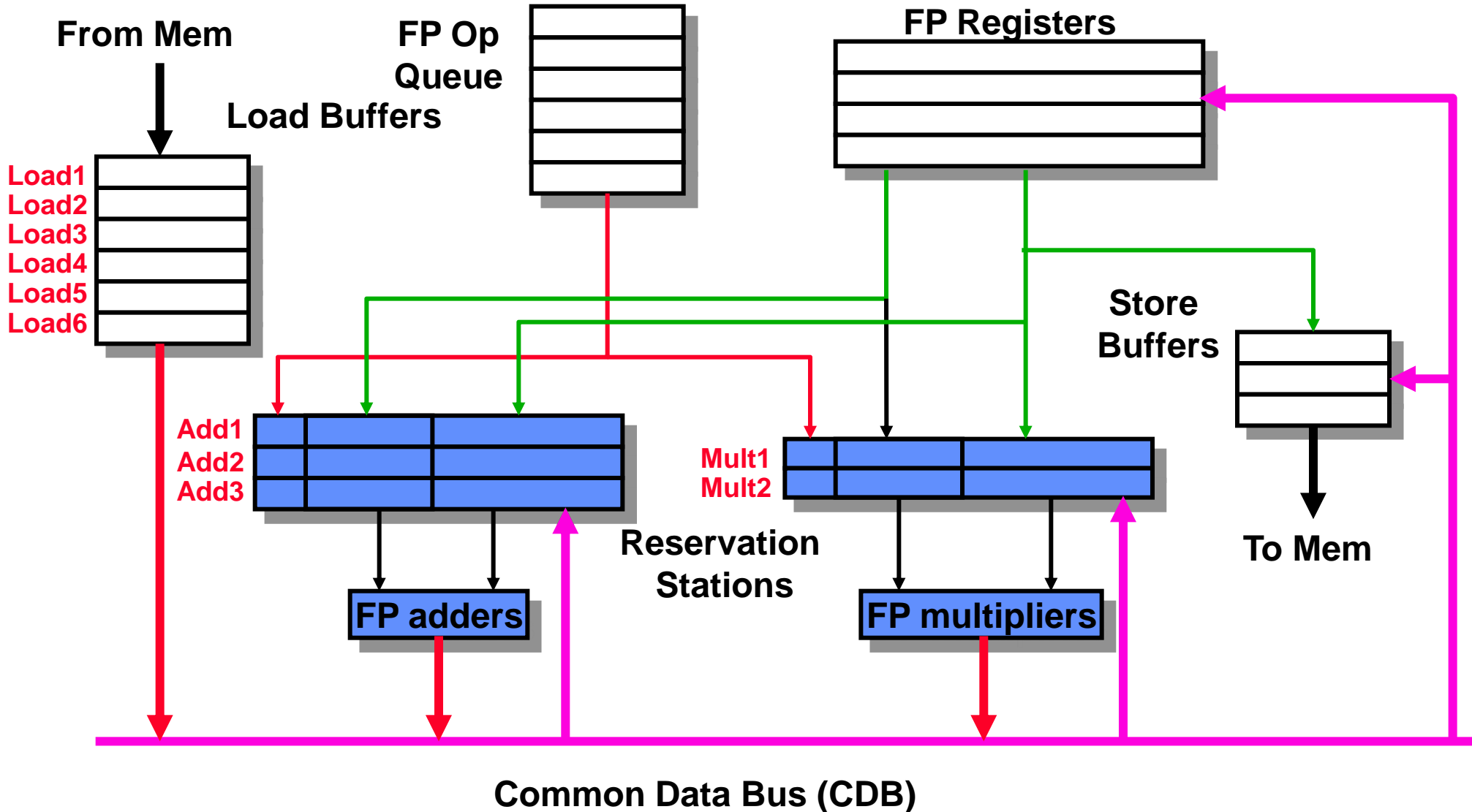
NASA's Space Flight Center in Greenbelt, Md, January 1968

Advanced Computer Architecture Chapter 3.15

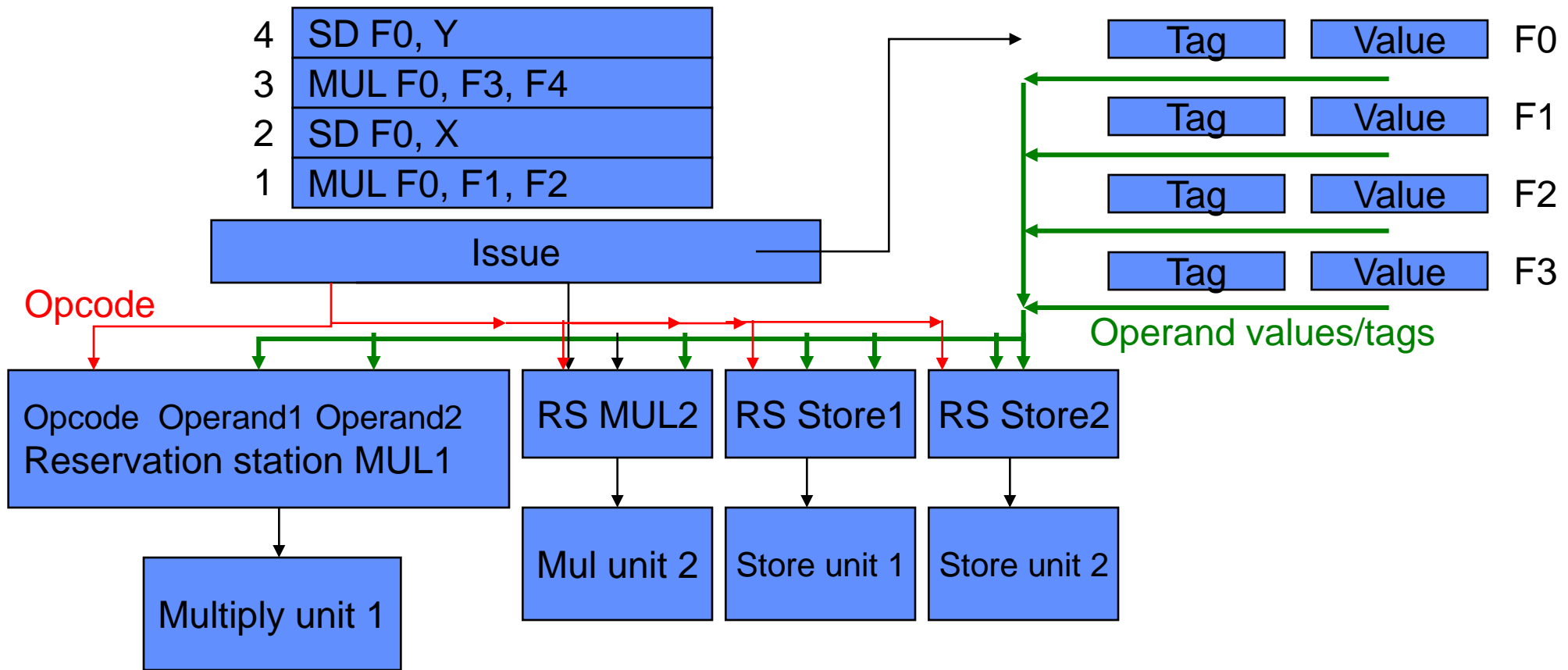
Tomasulo Algorithm

- 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 ;
 - 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
- Load and Stores treated as FUs with RSs as well
- Integer instructions can go past branches, allowing FP ops beyond basic block in FP queue

Tomasulo Organization



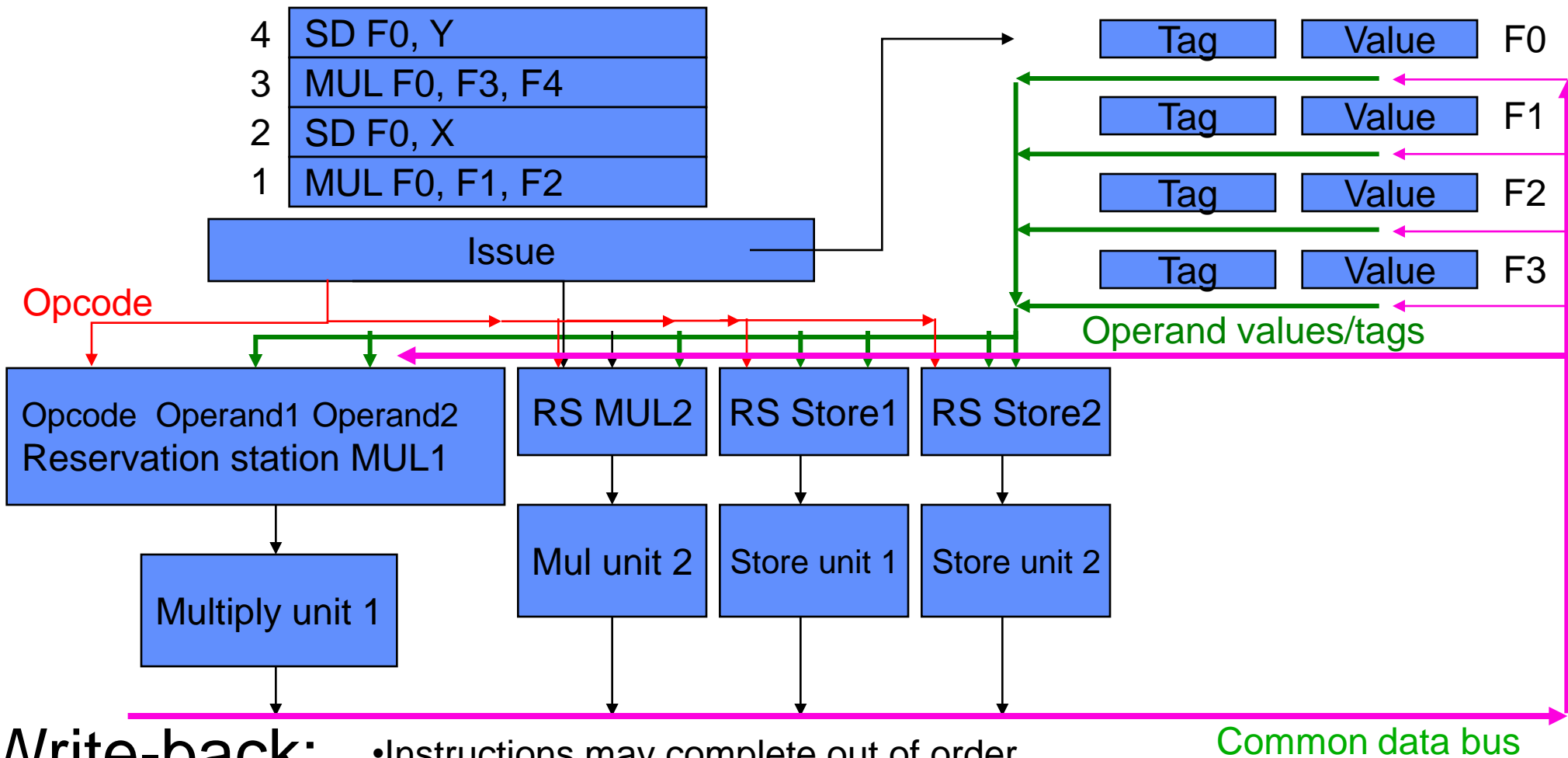
Tomasulo – closer look at instruction processing



Issue:

- Each instruction is issued in order
- Issue unit collects operands from the two instruction's source registers
- Result may be a value, or, if value will be computed by an uncompleted instruction, the tag of the RS to which it was issued.
- **When instruction 1 is issued, F0 is updated to get result from MUL1**
- **When instruction 3 is issued, F0 is updated to get result from MUL2**

Tomasulo – closer look at instruction processing



Write-back:

- Instructions may complete out of order
- Result is broadcast on CDB
- Carrying tag of RS to which instruction was originally issued
- All RSs and registers monitor CDB and collect value if tag matches
- Any RS which has both operands and whose FU is free fires.
- **When MUL1 completes result goes to store unit but not F0**

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: $Q_j, Q_k = 0 \Rightarrow$ 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 FI .pt. +,-; 10 for * ; 40 clks for /

360/91 pipeline

The IBM 360/91's pipeline:

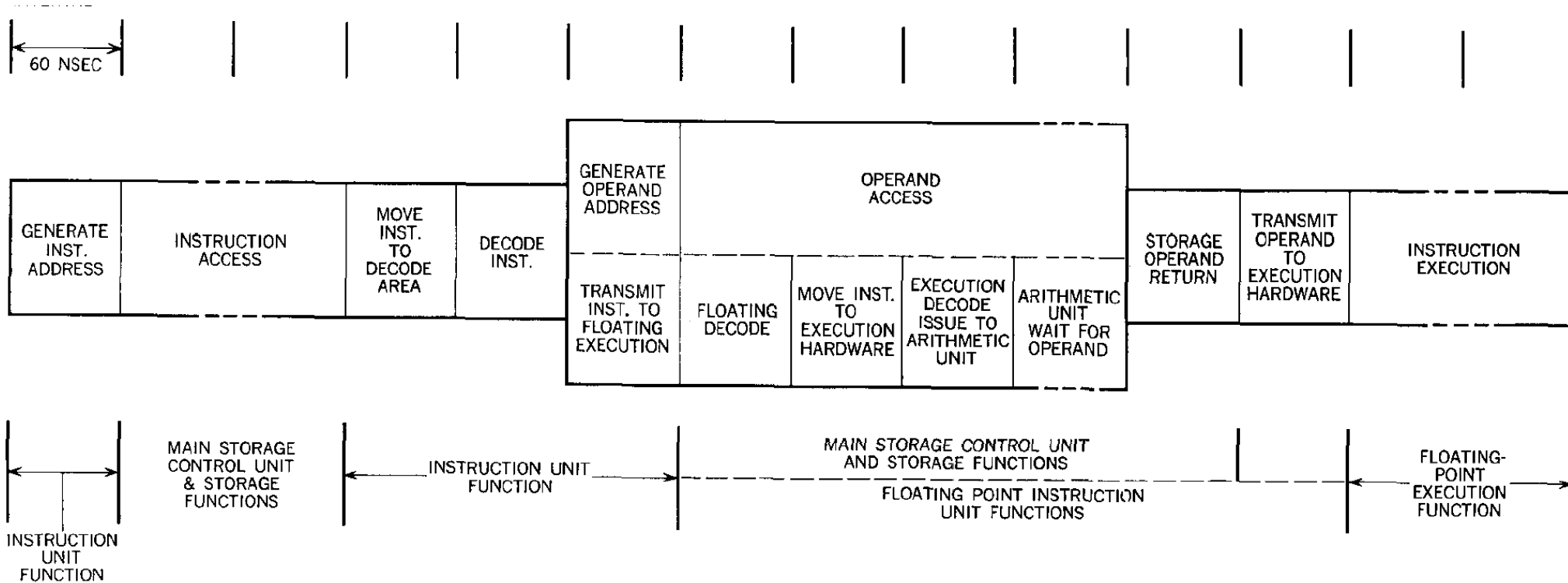


Figure 3 CPU “assembly-line stations required to accommodate a typical floating-point storage-to-register instruction.

11-12 circuit levels per pipeline stage, of 5-6ns each

CPU consists of three physical frames, each having dimensions 66" L X 15" D X 78" H

See: The IBM System/360 Model 91: Machine Philosophy and Instruction-Handling, by D. W. Anderson, F. J. Sparacio, R. M. Tomasulo. IBM J. R&D (1967), <http://www.research.ibm.com/journal/rd/111/ibmrd1101C.pdf>

Tomasulo Example

Instruction stream

Instruction status:

Instruction	<i>j</i>	<i>k</i>	<i>Issue</i>	<i>Exec</i>	<i>Write</i>
				<i>Comp</i>	<i>Result</i>
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:

FU count down

Time	Name	Busy	Op	<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					
	Mult2	No					

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

Register result status:

Clock
0
Clock cycle counter

	<i>F0</i>	<i>F2</i>	<i>F4</i>	<i>F6</i>	<i>F8</i>	<i>F10</i>	<i>F12</i>	...	<i>F30</i>
FU									

Tomasulo Example Cycle 1

Instruction status:

Instruction	<i>j</i>	<i>k</i>	<i>Issue</i>	<i>Exec</i>	<i>Write</i>	<i>Busy</i>	<i>Address</i>
LD	F6	34+	R2	1		Yes	34+R2
LD	F2	45+	R3			No	
MULTD	F0	F2	F4			No	
SUBD	F8	F6	F2				
DIVD	F10	F0	F6				
ADDD	F6	F8	F2				

Reservation Stations:

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

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>
1				Load1					

Tomasulo Example Cycle 2

Instruction status:

Instruction	<i>j</i>	<i>k</i>	<i>Issue</i>	<i>Exec</i>	<i>Write</i>
				<i>Comp</i>	<i>Result</i>
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		Load2			Load1				

Note: Can have multiple loads outstanding

Tomasulo Example Cycle 3

Instruction status:

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

Reservation Stations:

Time	Name	Busy	Op	S1 Vi	S2 Vk	RS Qi	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	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>Result</i>	Busy	Address
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:

Time	Name	Busy	Op	<i>S1</i>	<i>S2</i>	<i>RS</i>	<i>RS</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	FU	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>	Busy	Address
LD	F6	34+	R2	1	3	4	Load1
LD	F2	45+	R3	2	4	5	Load2
MULTD	F0	F2	F4	3			Load3
SUBD	F8	F6	F2	4			
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>
				<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	F0	F2	F4	F6	F8	F10	F12	...	F30
6	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 Comp</i>	<i>Write Result</i>	Busy	Address	
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	Op	<i>S1</i> <i>Vj</i>	<i>S2</i> <i>Vk</i>	<i>RS</i> <i>Qj</i>	<i>RS</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>	<i>Exec Write</i>			Busy	Address	
			<i>Issue</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:

Time	Name	Busy	Op	<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					
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>Result</i>	Busy	Address
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>
				<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>Result</i>	Busy	Address
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	Op	<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					
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	<i>j</i>	<i>k</i>	<i>Issue</i>	<i>Exec</i>	<i>Write</i>	Busy	Address
LD	F6	34+	R2	1	3	4	Load1
LD	F2	45+	R3	2	4	5	Load2
MULTD	F0	F2	F4	3			Load3
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	<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					
4	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>
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>Exec Write</i>			Busy	Address	
			<i>Issue</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:

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>
	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>
12	Mult1	M(A2)		(M-M+N)	(M-M)	Mult2			

Tomasulo Example Cycle 13

Instruction status:

Instruction	<i>j</i>	<i>k</i>	<i>Exec Write</i>			Busy	Address	
			<i>Issue</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>
13	<i>FU</i>	Mult1	M(A2)		(M-M+N)	(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>Result</i>	Busy	Address
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	Op	<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	Mult1	M(A2)		(M-M+N)	(M-M)	Mult2			

Tomasulo Example Cycle 15

Instruction status:

Instruction	<i>j</i>	<i>k</i>	<i>Issue</i>	<i>Exec</i>	<i>Write</i>	<i>Result</i>	Busy	Address
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	<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					
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	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>Result</i>	Busy	Address
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:

Time	Name	Busy	Op	<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	FU	M*F4	M(A2)		(M-M+N)	(M-M)	Mult2		

- Just waiting for Mult2 (DIVD) to complete

Skip a few cycles: Cycle 55

Instruction status:

Instruction	<i>j</i>	<i>k</i>	<i>Issue</i>	<i>Exec</i>	<i>Write</i>	<i>Result</i>	Busy	Address
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:

Time	Name	Busy	Op	<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>
55	<i>FU</i>	M*F4	M(A2)		(M-M+N)	(M-M)	Mult2		

Tomasulo Example Cycle 56

Instruction status:

Instruction	<i>j</i>	<i>k</i>	<i>Issue</i>	<i>Exec</i>	<i>Write</i>	<i>Result</i>	Busy	Address
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			
ADDD	F6	F8	F2	6	10	11		

Reservation Stations:

Time	Name	Busy	Op	<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+N)	(M-M)	Mult2		

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

Tomasulo Example Cycle 57

Instruction status:

Instruction	<i>j</i>	<i>k</i>	Issue	Exec Comp	Write Result	Busy	Address	
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	<i>S1</i> V _j	<i>S2</i> V _k	<i>RS</i> Q _j	<i>RS</i> Q _k
	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.

Tomasulo Drawbacks

Complexity

- delays of 360/91, MIPS 10000, Alpha 21264, IBM PPC 620
- 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		

- ▶ This time assume Multiply takes 4 clocks
- ▶ Assume 1st load takes 8 clocks
(L1 cache miss), 2nd load takes 1 clock (hit)
- ▶ To be clear, will show clocks for SUBI, BNEZ
 - ▶ Reality: integer instructions ahead of Fl. Pt. Instructions
- ▶ Show 2 iterations

Loop Example

Instruction status:

Iter- ation Count	ITER	Instruction	j	k	Exec Write		Fu			
					Issue	CompResult				
1	1	LD F0	0	R1			Load1	No		
1	1	MULTD F4	F0	F2			Load2	No		
1	1	SD F4	0	R1			Load3	No		
2	2	LD F0	0	R1			Store1	No		
2	2	MULTD F4	F0	F2			Store2	No		
2	2	SD F4	0	R1			Store3	No		

Reservation Stations:

Time	Name	Busy	Op	Vj	Vk	RS	
						Qj	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
```

Added Store Buffers

Instruction Loop

Register result status

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

Value of Register used for address, iteration control

Loop Example Cycle 1

Instruction status:

ITER	Instruction	<i>j</i>	<i>k</i>	Issue	Comp	Result	Busy	Addr	Fu
1	LD	F0	0	R1	1		Yes	80	
	Load2						No		
	Load3						No		
	Store1						No		
	Store2						No		
	Store3						No		

Reservation Stations:

Time	Name	Busy	Op	Vj	Vk	Qj	Qk	Code:
	Add1	No						LD F0 0 R1
	Add2	No						MULTD F4 F0 F2
	Add3	No						SD F4 0 R1
	Mult1	No						SUBI R1 R1 #8
	Mult2	No						BNEZ R1 Loop

Register result status

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

Loop Example Cycle 2

Instruction status:

ITER	Instruction	<i>j</i>	<i>k</i>	Issue	Comp	Result	Busy	Addr	Fu
1	LD	F0	0	R1	1		Yes	80	
1	MULTD	F4	F0	F2	2		No		
							No		
							No		
							No		
							No		
							No		

Reservation Stations:

Time	Name	Busy	Op	Vj	Vk	Qj	Qk	Code:
	Add1	No						LD F0 0 R1
	Add2	No						MULTD F4 F0 F2 ←
	Add3	No						SD F4 0 R1
	Mult1	Yes	Multd		R(F2)	Load1		SUBI R1 R1 #8
	Mult2	No						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	<i>j</i>	<i>k</i>	Issue	Comp	Result	Busy	Addr	Fu
1	LD	F0	0	R1	1		Yes	80	
1	MULTD	F4	F0	F2	2		No		
1	SD	F4	0	R1	3		No		
							Yes	80	Mult1
							No		
							No		

Reservation Stations:

Time	Name	Busy	Op	Vj	Vk	Qj	Qk	Code:
Add1		No						LD F0 0 R1
Add2		No						MULTD F4 F0 F2
Add3		No						SD F4 0 R1
Mult1	Yes	Multd		R(F2)	Load1			SUBI R1 R1 #8
Mult2	No							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	<i>j</i>	<i>k</i>	Issue	Comp	Result	Busy	Addr	Fu
1	LD	F0	0	R1	1		Yes	80	
1	MULTD	F4	F0	F2	2		No		
1	SD	F4	0	R1	3		No		
							Yes	80	Mult1
							No		
							No		

Reservation Stations:

Time	Name	Busy	Op	Vj	Vk	Qj	Qk	Code:
Add1		No						LD F0 0 R1
Add2		No						MULTD F4 F0 F2
Add3		No						SD F4 0 R1
Mult1		Yes	Multd		R(F2)	Load1		SUBI R1 R1 #8
Mult2		No						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		<i>j</i>	<i>k</i>	Issue	Comp	Result	Busy	Addr	Fu
1	LD	F0	0	R1	1			Yes	80	
1	MULTD	F4	F0	F2	2			No		
1	SD	F4	0	R1	3			No		
								Yes	80	Mult1
								No		
								No		

Reservation Stations:

Time	Name	Busy	Op	Vj	Vk	Qj	Qk	Code:			
	Add1	No						LD	F0	0	R1
	Add2	No						MULTD	F4	F0	F2
	Add3	No						SD	F4	0	R1
	Mult1	Yes	Multd		R(F2)	Load1		SUBI	R1	R1	#8
	Mult2	No						BNEZ	R1	Loop	

Register result status

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

And, BNEZ instruction (not in FP queue)

Loop Example Cycle 6

Instruction status:

ITER	Instruction		<i>j</i>	<i>k</i>	Issue	Comp	Result	Busy	Addr	Fu
1	LD	F0	0	R1	1			Yes	80	
1	MULTD	F4	F0	F2	2			Yes	72	
1	SD	F4	0	R1	3			No		
2	LD	F0	0	R1	6			Yes	80	Mult1
								No		
								No		

Reservation Stations:

Time	Name	Busy	Op	Vj	Vk	Qj	Qk	Code:
	Add1	No						LD F0 0 R1
	Add2	No						MULTD F4 F0 F2
	Add3	No						SD F4 0 R1
	Mult1	Yes	Multd		R(F2)	Load1		SUBI R1 R1 #8
	Mult2	No						BNEZ R1 Loop

Register result status

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

Notice that F0 never sees Load from location 80

Loop Example Cycle 7

Instruction status:

ITER	Instruction	<i>j</i>	<i>k</i>	Issue	Comp	Result	Busy	Addr	Fu
1	LD	F0	0	R1	1		Yes	80	
1	MULTD	F4	F0	F2	2		Yes	72	
1	SD	F4	0	R1	3		No		
2	LD	F0	0	R1	6		Yes	80	Mult1
2	MULTD	F4	F0	F2	7		No		
							No		

Reservation Stations:

Time	Name	Busy	Op	V _j	V _k	Q _j	Q _k	Code:
Add1		No						LD F0 0 R1
Add2		No						MULTD F4 F0 F2 ←
Add3		No						SD F4 0 R1
Mult1		Yes	Multd		R(F2)	Load1		SUBI R1 R1 #8
Mult2		Yes	Multd		R(F2)	Load2		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	<i>j</i>	<i>k</i>	Issue	Comp	Result	Busy	Addr	Fu
1	LD	F0	0	R1	1		Yes	80	
1	MULTD	F4	F0	F2	2		Yes	72	
1	SD	F4	0	R1	3		No		
2	LD	F0	0	R1	6		Yes	80	Mult1
2	MULTD	F4	F0	F2	7		Yes	72	Mult2
2	SD	F4	0	R1	8		No		

Reservation Stations:

Time	Name	Busy	Op	Vj	Vk	Qj	Qk	Code:
Add1		No						LD F0 0 R1
Add2		No						MULTD F4 F0 F2
Add3		No						SD F4 0 R1
Mult1		Yes	Multd		R(F2)	Load1		SUBI R1 R1 #8
Mult2		Yes	Multd		R(F2)	Load2		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	<i>j</i>	<i>k</i>	Issue	Comp	Result	Busy	Addr	<i>Fu</i>
1	LD	F0	0	R1	1	9	Yes	80	
1	MULTD	F4	F0	F2	2		Yes	72	
1	SD	F4	0	R1	3		No		
2	LD	F0	0	R1	6		Yes	80	Mult1
2	MULTD	F4	F0	F2	7		Yes	72	Mult2
2	SD	F4	0	R1	8		No		

Reservation Stations:

Time	Name	Busy	Op	V _j	V _k	Q _j	Q _k	Code:
	Add1	No						LD F0 0 R1
	Add2	No						MULTD F4 F0 F2
	Add3	No						SD F4 0 R1
	Mult1	Yes	Multd		R(F2)	Load1		SUBI R1 R1 #8 ←
	Mult2	Yes	Multd		R(F2)	Load2		BNEZ R1 Loop

Register result status

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

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

Loop Example Cycle 10

Instruction status:

ITER	Instruction	<i>j</i>	<i>k</i>	Issue	Comp	Result	Busy	Addr	Fu	
1	LD	F0	0	R1	1	9	10	Load1	No	
1	MULTD	F4	F0	F2	2		Load2	Yes	72	
1	SD	F4	0	R1	3		Load3	No		
2	LD	F0	0	R1	6	10	Store1	Yes	80	Mult1
2	MULTD	F4	F0	F2	7		Store2	Yes	72	Mult2
2	SD	F4	0	R1	8		Store3	No		

Reservation Stations:

Time	Name	Busy	Op	Vj	Vk	Qj	Qk	Code:
	Add1	No						LD
	Add2	No						MULTD
	Add3	No						SD
4	Mult1	Yes	Multd	M[80]	R(F2)			SUBI
	Mult2	Yes	Multd		R(F2)	Load2		BNEZ

Register result status

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

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

Loop Example Cycle 11

Instruction status:

ITER	Instruction	j	k	Exec Write			Busy	Addr	Fu
				Issue	Comp	Result			
1	LD	F0	0	R1	1	9	10	Load1	No
1	MULTD	F4	F0	F2	2			Load2	No
1	SD	F4	0	R1	3			Load3	Yes 64
2	LD	F0	0	R1	6	10	11	Store1	Yes 80 Mult1
2	MULTD	F4	F0	F2	7			Store2	Yes 72 Mult2
2	SD	F4	0	R1	8			Store3	No

Reservation Stations:

Time	Name	Busy	Op	Vj	Vk	Qj	Qk	Code:
	Add1	No						LD F0 0 R1
	Add2	No						MULTD F4 F0 F2
	Add3	No						SD F4 0 R1
3	Mult1	Yes	Multd	M[80]	R(F2)			SUBI R1 R1 #8
4	Mult2	Yes	Multd	M[72]	R(F2)			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	<i>j</i>	<i>k</i>	Issue	Comp	Result	Exec	Write	Busy	Addr	Fu
1	LD	F0	0	R1	1	9	10	Load1	No		
1	MULTD	F4	F0	F2	2			Load2	No		
1	SD	F4	0	R1	3			Load3	Yes	64	
2	LD	F0	0	R1	6	10	11	Store1	Yes	80	Mult1
2	MULTD	F4	F0	F2	7			Store2	Yes	72	Mult2
2	SD	F4	0	R1	8			Store3	No		

Reservation Stations:

Time	Name	Busy	Op	Vj	Vk	Qj	Qk	Code:
	Add1	No						LD F0 0 R1
	Add2	No						MULTD F4 F0 F2 ←
	Add3	No						SD F4 0 R1
2	Mult1	Yes	Multd	M[80]	R(F2)			SUBI R1 R1 #8
3	Mult2	Yes	Multd	M[72]	R(F2)			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	<i>j</i>	<i>k</i>	Issue	Comp	Result	Exec	Write	Busy	Addr	Fu
1	LD	F0	0	R1	1	9	10	Load1	No		
1	MULTD	F4	F0	F2	2			Load2	No		
1	SD	F4	0	R1	3			Load3	Yes	64	
2	LD	F0	0	R1	6	10	11	Store1	Yes	80	Mult1
2	MULTD	F4	F0	F2	7			Store2	Yes	72	Mult2
2	SD	F4	0	R1	8			Store3	No		

Reservation Stations:

Time	Name	Busy	Op	Vj	Vk	Qj	Qk	Code:
	Add1	No						LD F0 0 R1
	Add2	No						MULTD F4 F0 F2 ←
	Add3	No						SD F4 0 R1
1	Mult1	Yes	Multd	M[80]	R(F2)			SUBI R1 R1 #8
2	Mult2	Yes	Multd	M[72]	R(F2)			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	<i>j</i>	<i>k</i>	Issue	Comp	Result	Exec	Write	Busy	Addr	Fu
1	LD	F0	0	R1	1	9	10		No		
1	MULTD	F4	F0	F2	2	14			No		
1	SD	F4	0	R1	3				Yes	64	
2	LD	F0	0	R1	6	10	11		Yes	80	Mult1
2	MULTD	F4	F0	F2	7				Yes	72	Mult2
2	SD	F4	0	R1	8				No		

Reservation Stations:

Time	Name	Busy	Op	V _j	V _k	Q _j	Q _k	Code:
	Add1	No						LD F0 0 R1
	Add2	No						MULTD F4 F0 F2 ←
	Add3	No						SD F4 0 R1
0	Mult1	Yes	Multd	M[80]	R(F2)			SUBI R1 R1 #8
1	Mult2	Yes	Multd	M[72]	R(F2)			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	<i>j</i>	<i>k</i>	Issue	Comp	Result	Exec	Write	Busy	Addr	Fu
1	LD	F0	0	R1	1	9	10	Load1	No		
1	MULTD	F4	F0	F2	2	14	15	Load2	No		
1	SD	F4	0	R1	3			Load3	Yes	64	
2	LD	F0	0	R1	6	10	11	Store1	Yes	80	[80]*R2
2	MULTD	F4	F0	F2	7	15		Store2	Yes	72	Mult2
2	SD	F4	0	R1	8			Store3	No		

Reservation Stations:

Time	Name	Busy	Op	Vj	Vk	Qj	Qk	Code:
	Add1	No						LD F0 0 R1
	Add2	No						MULTD F4 F0 F2 ←
	Add3	No						SD F4 0 R1
	Mult1	No						SUBI R1 R1 #8
0	Mult2	Yes	Multd	M[72]	R(F2)			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	Exec Write			Busy	Addr	Fu		
				Issue	Comp	Result					
1	LD	F0	0	R1	1	9	10	Load1	No		
1	MULTD	F4	F0	F2	2	14	15	Load2	No		
1	SD	F4	0	R1	3			Load3	Yes	64	
2	LD	F0	0	R1	6	10	11	Store1	Yes	80	[80]*R2
2	MULTD	F4	F0	F2	7	15	16	Store2	Yes	72	[72]*R2
2	SD	F4	0	R1	8			Store3	No		

Reservation Stations:

Time	Name	Busy	Op	Vj	Vk	Qj	Qk	Code:			
									S1	S2	RS
	Add1	No						LD	F0	0	R1
	Add2	No						MULTD	F4	F0	F2
	Add3	No						SD	F4	0	R1
4	Mult1	Yes	Multd		R(F2)	Load3		SUBI	R1	R1	#8
	Mult2	No						BNEZ	R1	Loop	

Register result status

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

Loop Example Cycle 17

Instruction status:

ITER	Instruction	<i>j</i>	<i>k</i>	Issue	Comp	Result	Exec	Write	Busy	Addr	Fu
1	LD	F0	0	R1	1	9	10	Load1	No		
1	MULTD	F4	F0	F2	2	14	15	Load2	No		
1	SD	F4	0	R1	3			Load3	Yes	64	
2	LD	F0	0	R1	6	10	11	Store1	Yes	80	[80]*R2
2	MULTD	F4	F0	F2	7	15	16	Store2	Yes	72	[72]*R2
2	SD	F4	0	R1	8			Store3	Yes	64	Mult1

Reservation Stations:

Time	Name	Busy	Op	Vj	Vk	Qj	Qk	Code:
	Add1	No						LD F0 0 R1
	Add2	No						MULTD F4 F0 F2
	Add3	No						SD F4 0 R1
	Mult1	Yes	Multd		R(F2)	Load3		SUBI R1 R1 #8
	Mult2	No						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	<i>j</i>	<i>k</i>	Issue	Comp	Result	Exec	Write	Busy	Addr	Fu
1	LD	F0	0	R1	1	9	10	Load1	No		
1	MULTD	F4	F0	F2	2	14	15	Load2	No		
1	SD	F4	0	R1	3	18		Load3	Yes	64	
2	LD	F0	0	R1	6	10	11	Store1	Yes	80	[80]*R2
2	MULTD	F4	F0	F2	7	15	16	Store2	Yes	72	[72]*R2
2	SD	F4	0	R1	8			Store3	Yes	64	Mult1

Reservation Stations:

Time	Name	Busy	Op	Vj	Vk	Qj	Qk	Code:
	Add1	No						LD F0 0 R1
	Add2	No						MULTD F4 F0 F2
	Add3	No						SD F4 0 R1
	Mult1	Yes	Multd		R(F2)	Load3		SUBI R1 R1 #8 ←
	Mult2	No						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	<i>j</i>	<i>k</i>	Issue	Comp	Result	Exec	Write	Busy	Addr	Fu
1	LD	F0	0	R1	1	9	10	Load1	No		
1	MULTD	F4	F0	F2	2	14	15	Load2	No		
1	SD	F4	0	R1	3	18	19	Load3	Yes	64	
2	LD	F0	0	R1	6	10	11	Store1	No		
2	MULTD	F4	F0	F2	7	15	16	Store2	Yes	72	[72]*R2
2	SD	F4	0	R1	8	19		Store3	Yes	64	Mult1

Reservation Stations:

Time	Name	Busy	Op	Vj	Vk	Qj	Qk	Code:
Add1	No							LD F0 0 R1
Add2	No							MULTD F4 F0 F2
Add3	No							SD F4 0 R1
Mult1	Yes	Multd			R(F2)	Load3		SUBI R1 R1 #8
Mult2	No							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	<i>j</i>	<i>k</i>	Issue	Comp	Result	Exec	Write	Busy	Addr	Fu
1	LD	F0	0	R1	1	9	10	Load1	Yes	56	
1	MULTD	F4	F0	F2	2	14	15	Load2	No		
1	SD	F4	0	R1	3	18	19	Load3	Yes	64	
2	LD	F0	0	R1	6	10	11	Store1	No		
2	MULTD	F4	F0	F2	7	15	16	Store2	No		
2	SD	F4	0	R1	8	19	20	Store3	Yes	64	Mult1

Reservation Stations:

Time	Name	Busy	Op	Vj	Vk	Qj	Qk	Code:
	Add1	No						LD F0 0 R1 ←
	Add2	No						MULTD F4 F0 F2
	Add3	No						SD F4 0 R1
	Mult1	Yes	Multd		R(F2)	Load3		SUBI R1 R1 #8
	Mult2	No						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.

Tomasulo's scheme offers two major advantages

(1) the 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) the elimination of stalls for WAW and WAR hazards

What about Precise Interrupts?

▶ **Tomasulo had:**

In-order issue, out-of-order execution, and out-of-order completion

▶ **Need to “fix” the out-of-order completion aspect so that we can find precise breakpoint in instruction stream.**

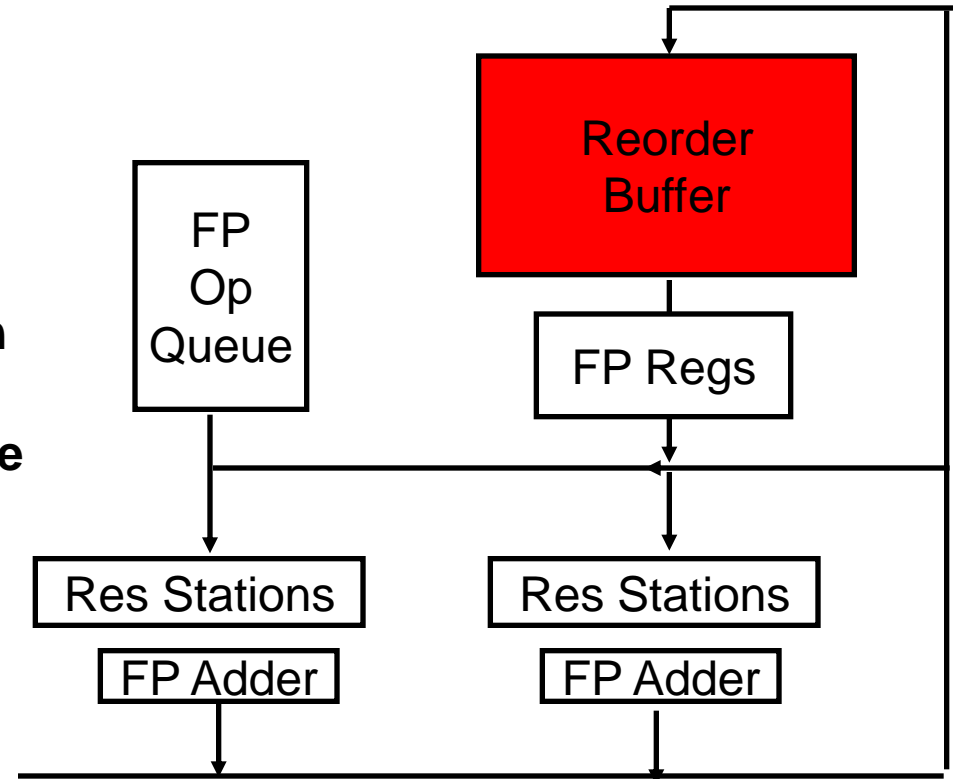
Relationship between precise interrupts and speculation:

- ▶ Speculation is a form of guessing.
- ▶ Important for branch prediction:
 - ▶ Need to “take our best shot” at predicting branch direction.
- ▶ If we speculate and are wrong, need to back up and restart execution at 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*

HW support for precise interrupts

Need HW buffer for results of uncommitted instructions:
reorder buffer

- ➔ 3 fields: instr, destination, value
- ➔ Use reorder buffer number instead of reservation station when execution completes
- ➔ Supplies operands between execution complete & commit
- ➔ (Reorder buffer can be operand source => more registers like RS)
- ➔ Instructions commit
- ➔ Once instruction commits, result is put into register
- ➔ As a result, easy to undo speculated instructions on mispredicted branches or exceptions



Four Steps of Speculative Tomasulo Algorithm

1. Issue—get instruction from FP Op Queue

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

2. Execution—operate on operands (EX)

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

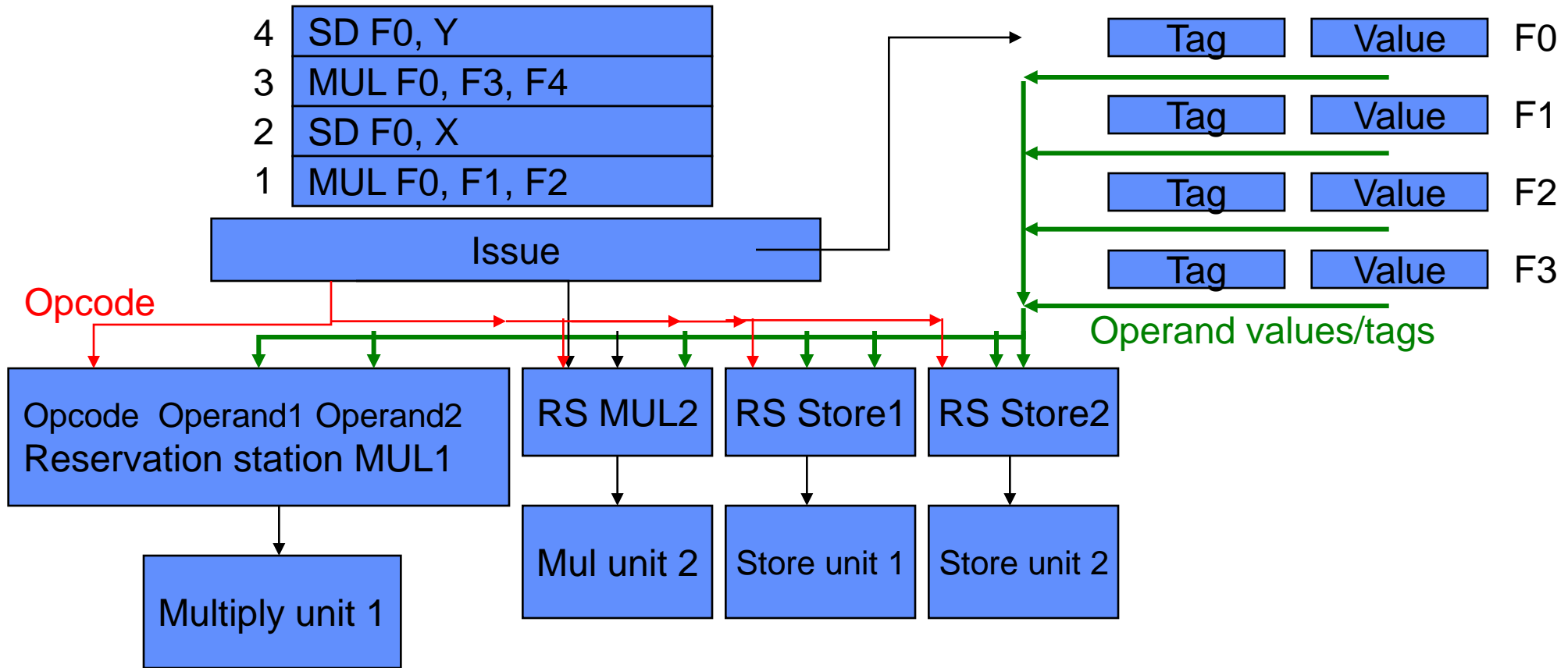
3. Write result—finish execution (WB)

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

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 without Re-order Buffer

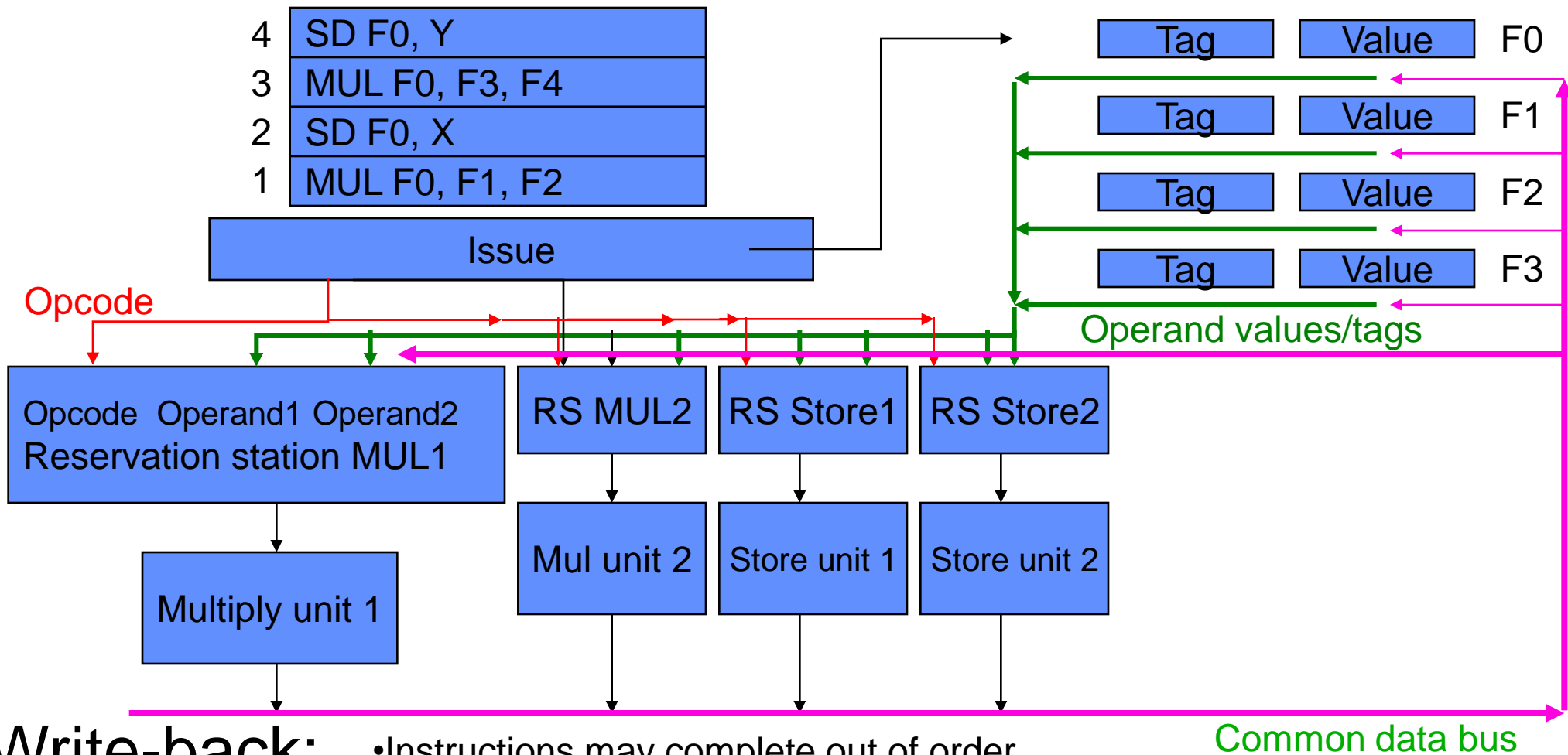


Issue:

- Each instruction is issued in order
- Issue unit collects operands from the two instruction's source registers
- Result may be a value, or, if value will be computed by an uncompleted instruction, the tag of the RS to which it was issued.
- **When instruction 1 is issued, F0 is updated to get result from MUL1**
- **When instruction 3 is issued, F0 is updated to get result from MUL2**

(Copy of earlier slide)

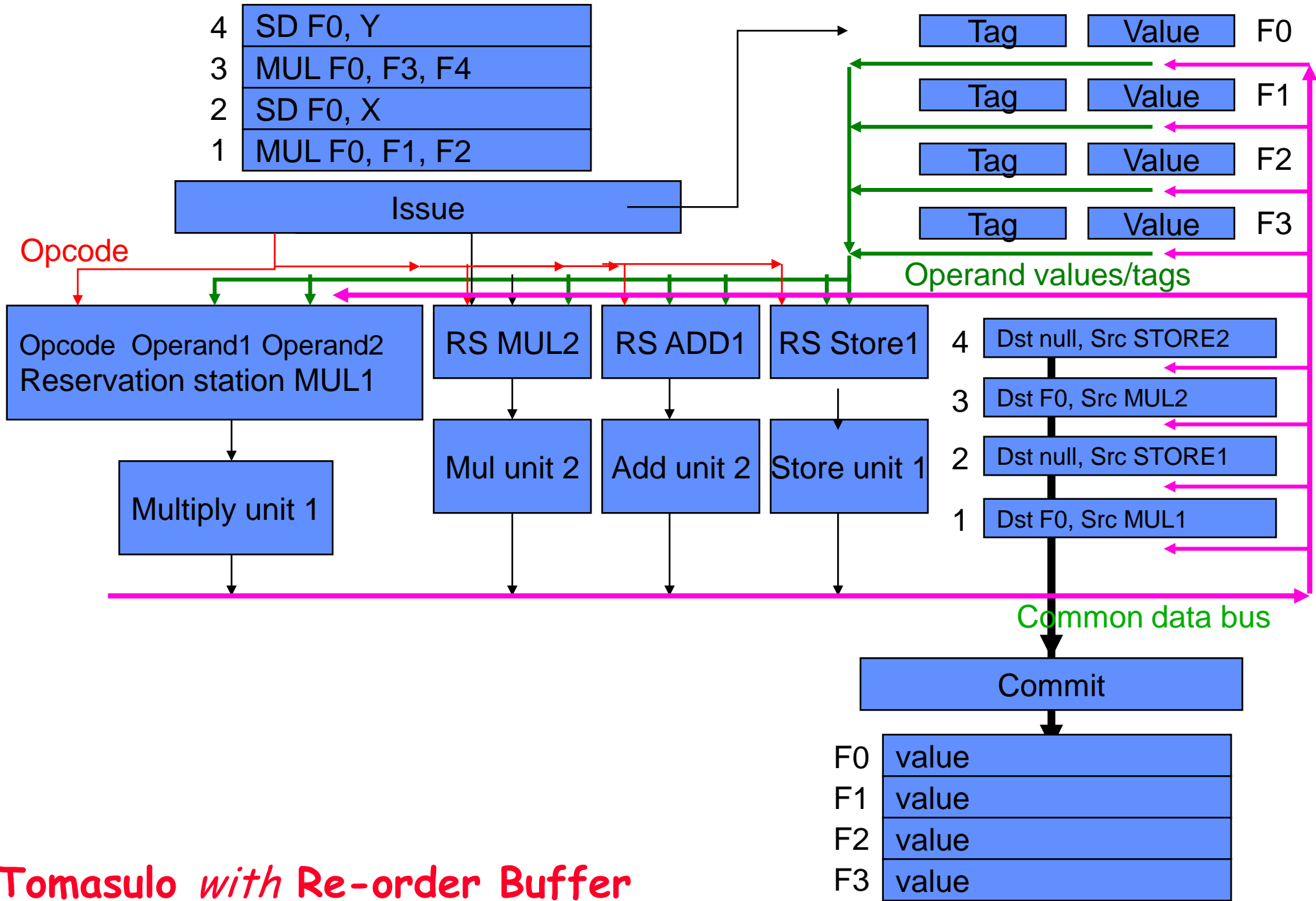
Tomasulo without Re-order Buffer



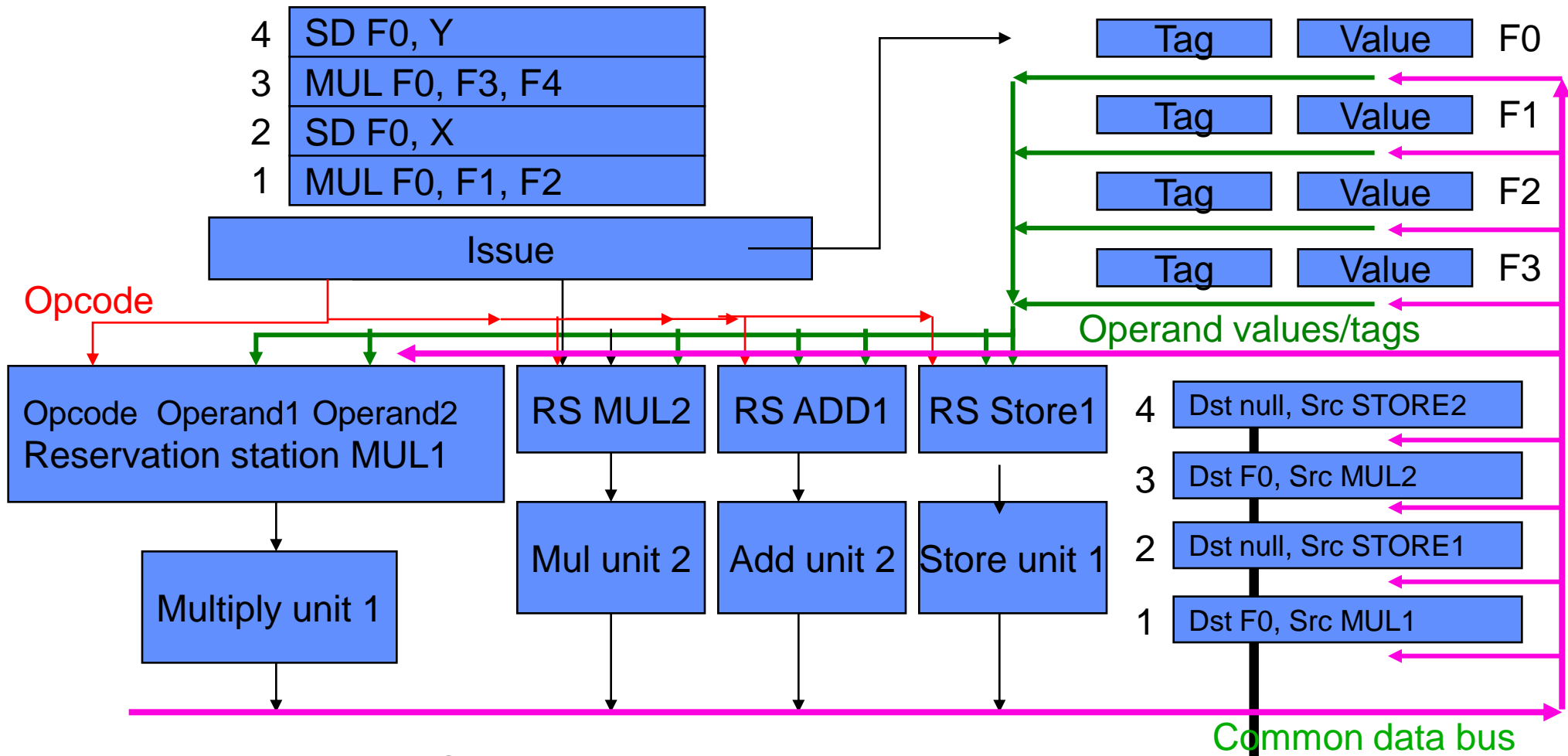
Write-back:

- Instructions may complete out of order
- Result is broadcast on CDB
- Carrying tag of RS to which instruction was originally issued
- All RSs and registers monitor CDB and collect value if tag matches
- Any RS which has both operands and whose FU is free fires.
- **When MUL1 completes result goes to store unit but not F0**

(Copy of earlier slide)



Tomasulo with Re-order Buffer



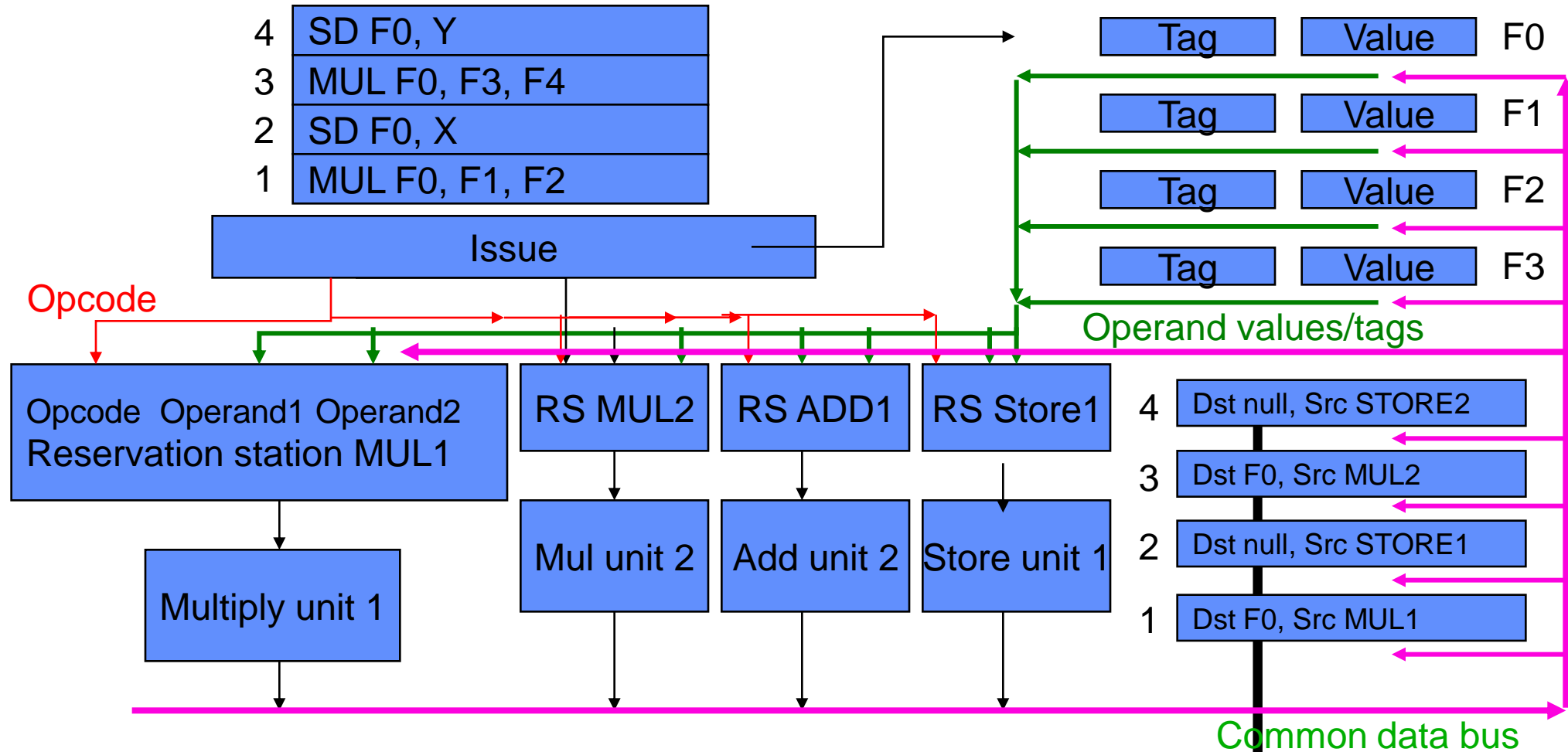
Issue: •As before, but ROB entry is also allocated

•ROB entry for each instruction

•Holds destination register + value/tag for where it will come from

Tomasulo with Re-order Buffer

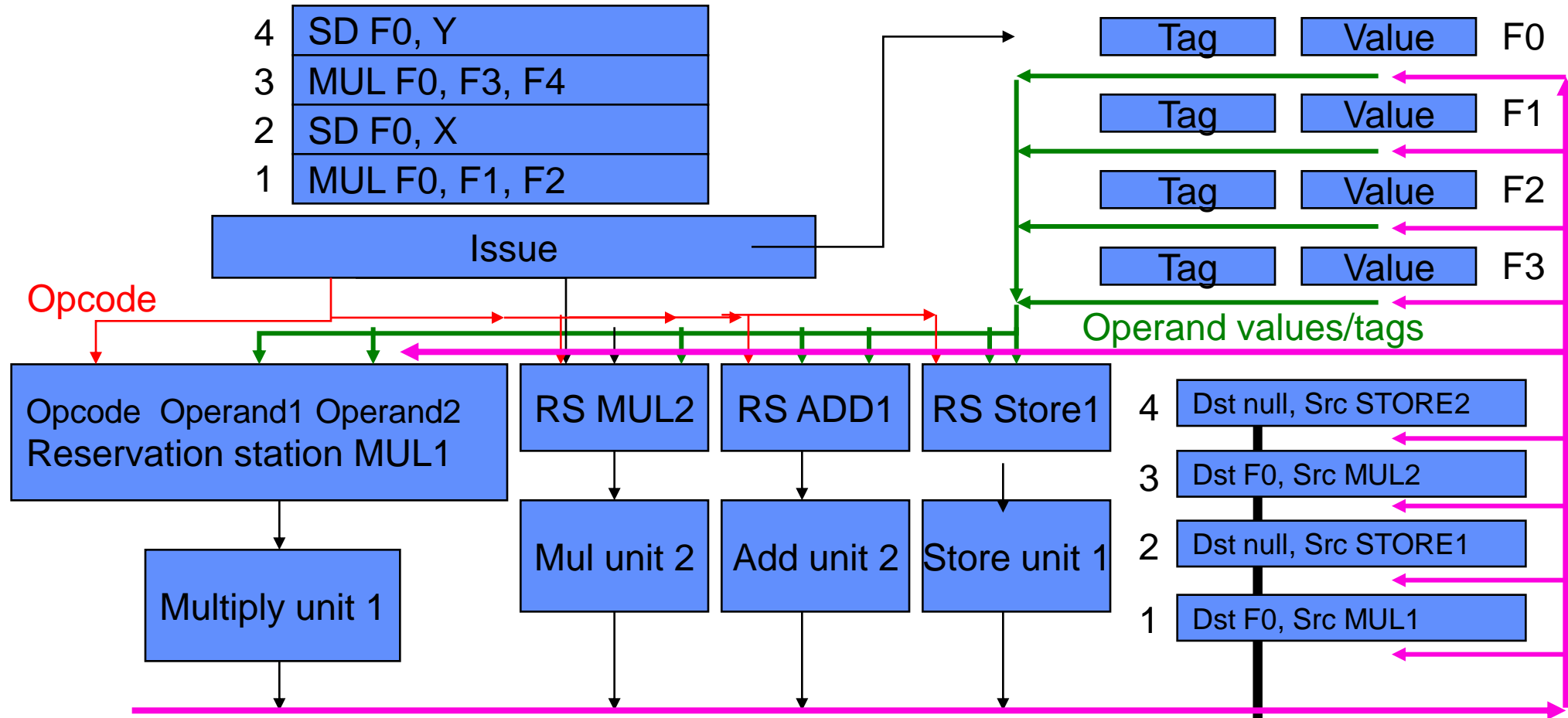
F0	value
F1	value
F2	value
F3	value



Write Back:

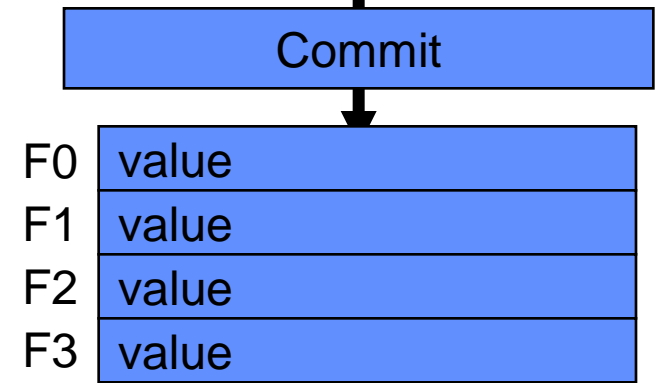
- As before, but ROB entry with matching tag also updated
- ROB entry for instruction 1 holds value for F0
- ROB entry for instruction 3 holds another value for F0

Tomasulo with Re-order Buffer



Commit:

- Commit unit processes ROB entries in issue order
- Each instruction waits in turn and commits when its operands are completed
- Committed registers updated with values from ROB
- F0 is updated first with result from MUL1 then result from MUL2



```

4 SD F0, Y
3 MUL F0, F3, F4
2 SD F0, X
1 MUL F0, F1, F2

```

Tag	Value	F0
Tag	Value	F1
Tag	Value	F2
Tag	Value	F3

Operand values/tags

Issue

Opcode

Opcode Operand1 Operand2
Reservation station MUL1

RS MUL2

RS ADD1

RS Store1

4 Dst null, Src STORE2

3 Dst F0, Src MUL2

2 Dst null, Src STORE1

1 Dst F0, Src MUL1

Multiply unit 1

Mul unit 2

Add unit 2

Store unit 1

Common data bus

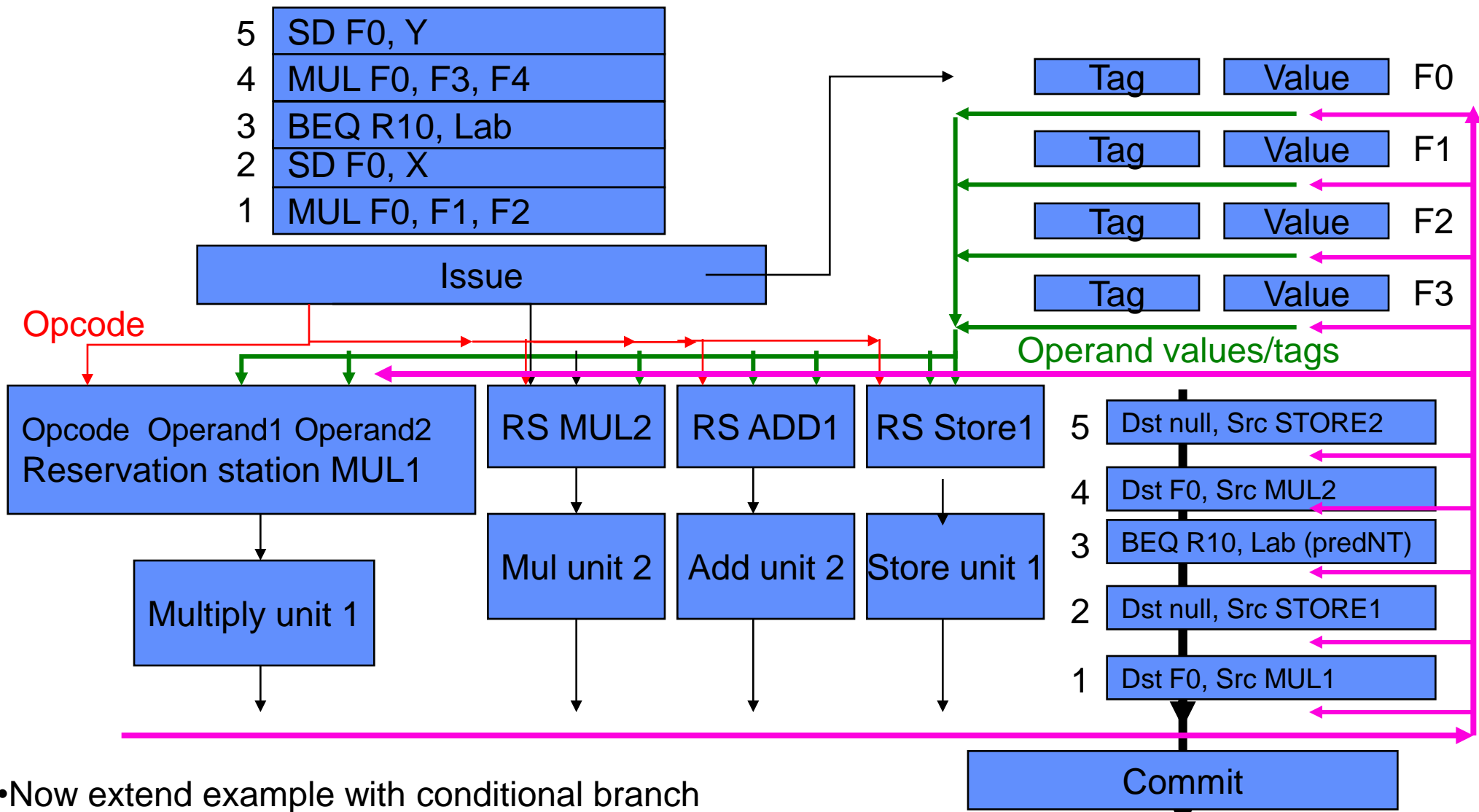
Commit

F0	value
F1	value
F2	value
F3	value

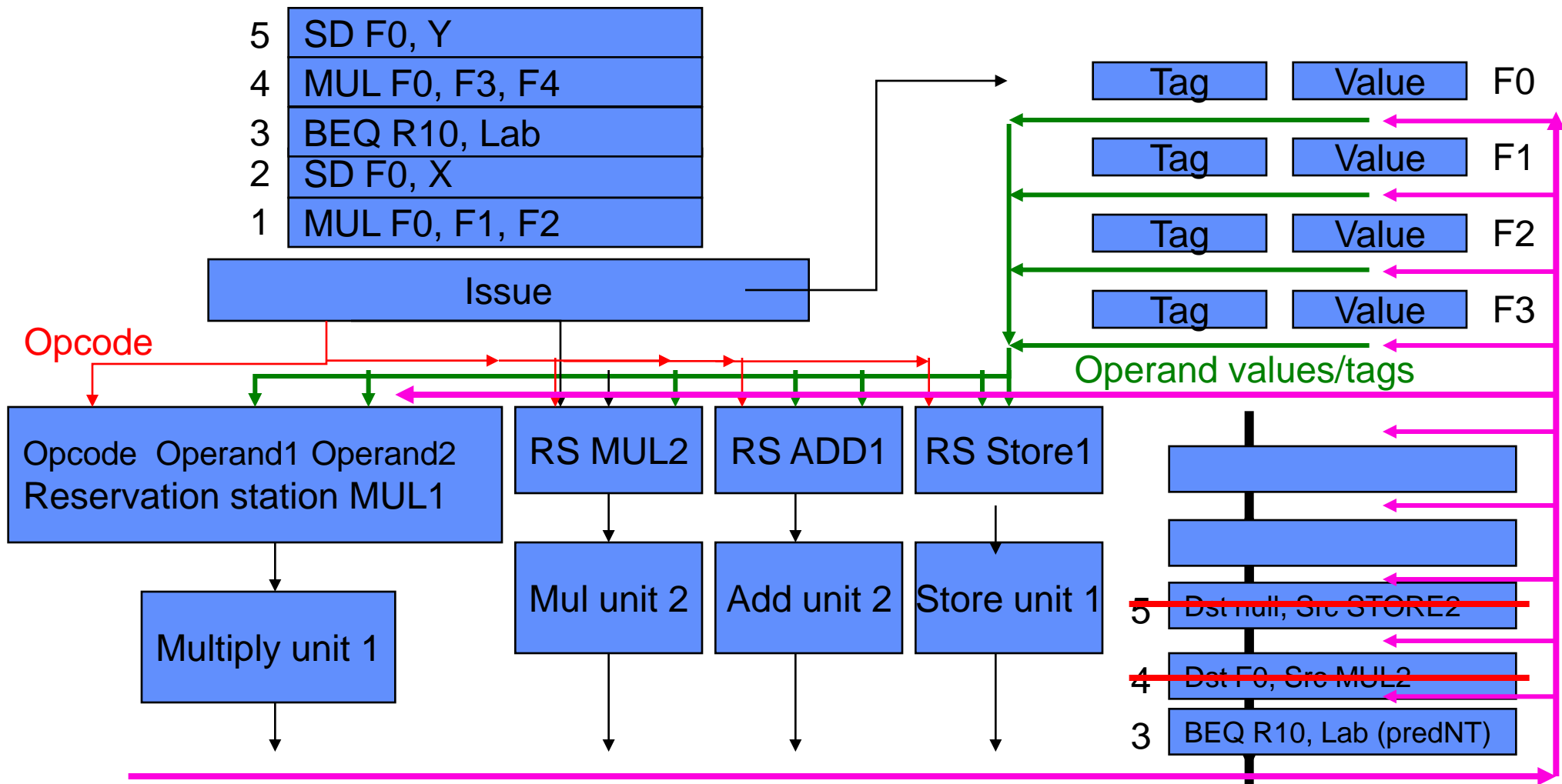
Issue-side registers
(updated speculatively)

Commit-side registers
(updated when speculation resolved)

Tomasulo with Re-order Buffer

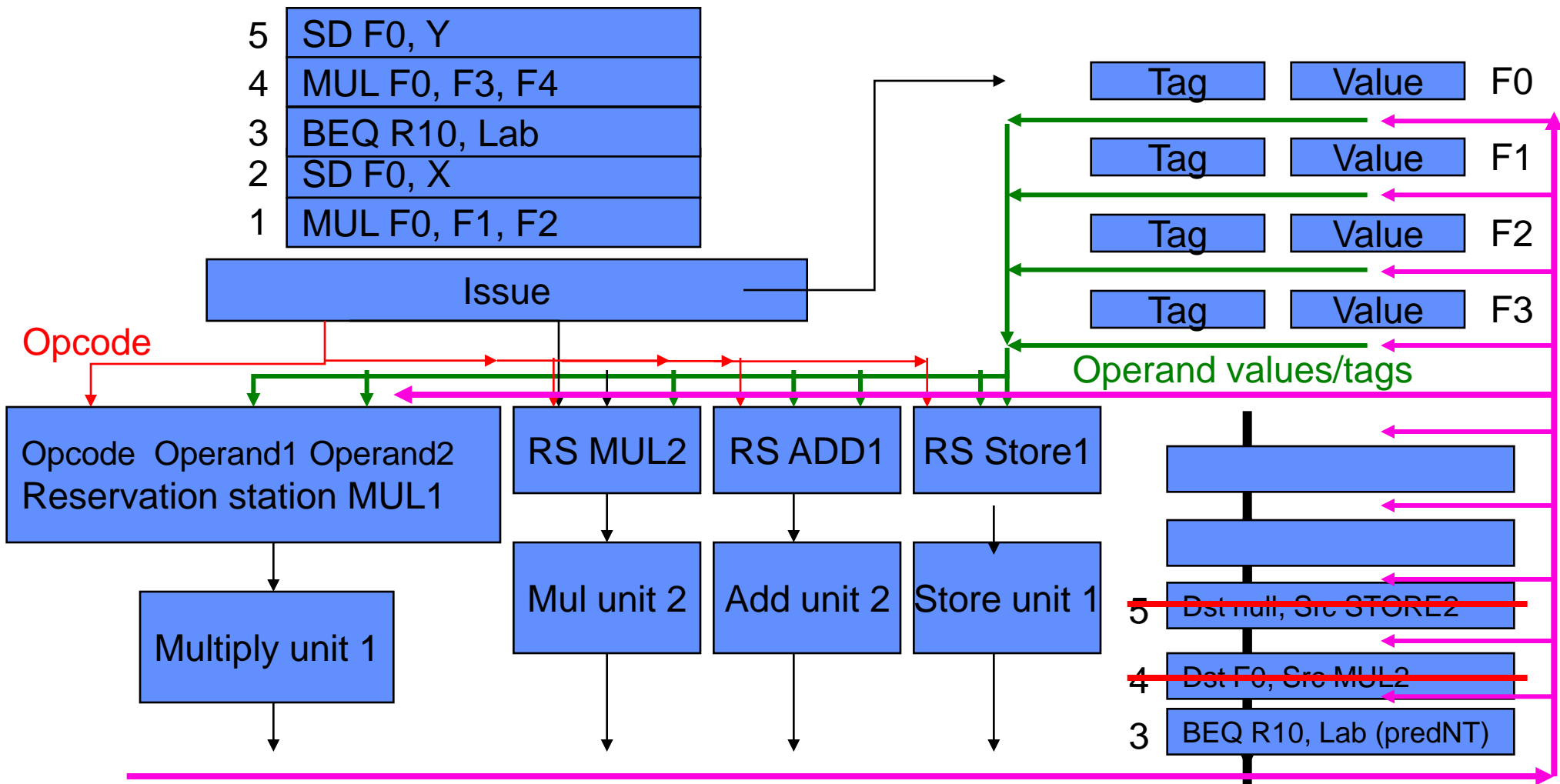


- Now extend example with conditional branch
- Assume predicted Not Taken
- When BEQ reaches head of commit queue, all instructions which have been issued but have not yet committed are erroneous



- **Misprediction:** all ROB entries are trashed
- Issue-side registers reset from commit-side registers
- Correct branch target instruction fetched and issued

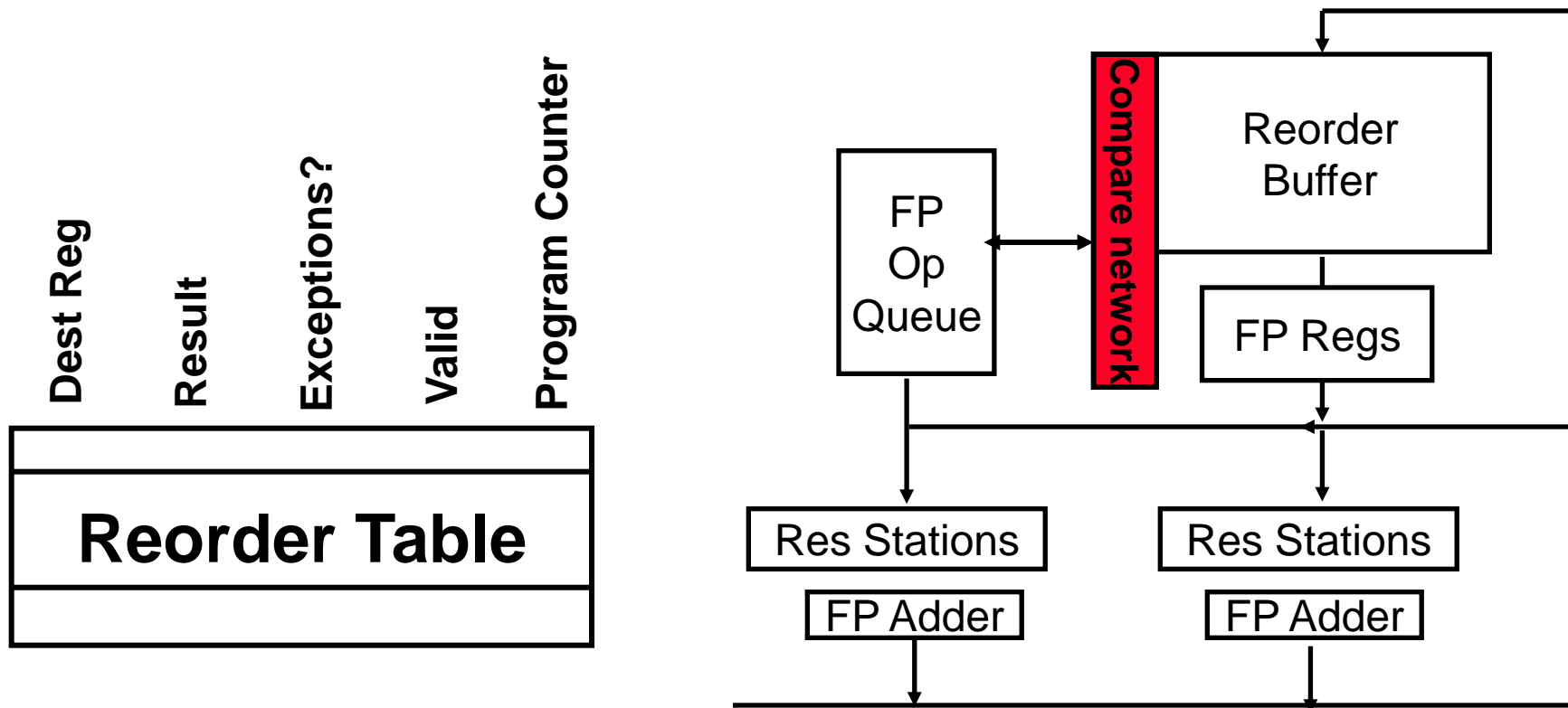
F0	Value from MUL1
F1	value
F2	value
F3	value



- Committed F0 holds value from first MUL
- RS of uncompleted speculatively-executed instruction cannot be re-used until its FU (eg MUL2) completes

F0	Value from MUL1
F1	value
F2	value
F3	value

What are the hardware complexities with reorder buffer (ROB)?



➤ How do you find the latest version of a register?

- Looks like we need associative comparison network
- Could use future file or just use the register result status buffer to track which specific reorder buffer has received the value

➤ Need as many ports on ROB as register file

Some subtleties...

- ▶ **It's vital to reduce the branch misprediction penalty. Does the Tomasulo+ROB scheme described here roll-back as soon as the branch is found to be mispredicted?**
- ▶ **Stores are buffered in the ROB, and committed only when the instruction is committed. A load can be issued while several stores (perhaps to the same address) are uncommitted. We need to make sure the load gets the right data.**
- ▶ **What if a second conditional branch is encountered, before the outcome of the first is resolved?**
- ▶ **This discussion has assumed a single-issue machine. How can these ideas be extended to allow multiple instructions to be issued per cycle?**
 - ▶ **Issue**
 - ▶ **Monitoring CDBs for completion**
 - ▶ **Handling multiple commits per cycle**

Tomasulo + ROB: Summary

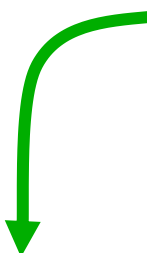
- ▶ **Reservations stations: *implicit register renaming* to larger set of registers + buffering source operands**
 - ▶ Prevents registers as bottleneck
 - ▶ Avoids WAR, WAW hazards of Scoreboard (see textbook)
 - ▶ Allows loop unrolling in HW
- ▶ **Not limited to basic blocks (integer units gets ahead, beyond branches)**
- ▶ **Today, helps cache misses as well**
 - ▶ Don't stall for L1 Data cache miss (insufficient ILP for L2 miss?)
- ▶ **Lasting Contributions**
 - ▶ Dynamic scheduling
 - ▶ Register renaming
 - ▶ Load/store disambiguation

Out-of-order processing - Four instructions per cycle

Naive implementation (roughly from `cc -S`):

```
Example:  
void f() {  
    int i, a;  
    for (i=1; i<=1000000000;  
         i++)  
        a = a+i;  
}
```

X86 code (slightly tidied
but without register
allocation)



```
movl $1,-4(%ebp)  
jmp .L4  
.L5  
    movl -4(%ebp),%eax  
    addl %eax,-8(%ebp)  
    incl -4(%ebp)  
.L4:  
    cmpl $1000000000,-4(%ebp)  
    jle .L5
```

Unoptimised:

```
movl $1,-4(%ebp)
jmp .L4
.L5
movl -4(%ebp),%eax
addl %eax,-8(%ebp)
incl -4(%ebp)
.L4:
cmpl $1000000000,-4(%ebp)
jle .L5
```

5 instructions in the loop

Execution time on 2.13GHz Intel
Core2Duo: 3.87 seconds (3.87
nanoseconds/iteration, 8.24 cycles)

Optimised:

```
movl $1,%edx
.L6:
addl %edx,%eax
incl %edx
cmpl $1000000000,%edx
jle .L6
```

4 instructions in the loop, no
references to main memory

Execution time on 2.13GHz Intel
Core2Duo: 0.48 seconds (0.48
nanoseconds/iteration, 1.02 cycles)

Time per instruction fell: 0.77 nanoseconds to 0.12

Optimised code runs at four instructions per cycle

Papers:

- Instruction issue logic for high-performance, interruptable pipelined processors. G. S. Sohi, S. Vajapeyam. International Conference on Computer Architecture, 1987 (<http://doi.acm.org/10.1145/30350.30354>)
- Towards Kilo-instruction processors. Cristal, Santana, Valero, Martinez ACM Trans. Architecture and Code Optimization (<http://doi.acm.org/10.1145/1044823.1044825>)

Animations:

- SATSim Simplescalar
 - <http://www.ece.gatech.edu/research/pica/SATSim/satsim.html>
- WebHase Tomasulo model:
 - www.dcs.ed.ac.uk/home/hase/webhase/demo/tomasulo.html
- Other WebHase animations – simple pipeline, Scoreboarding etc:
 - <http://www.icsa.informatics.ed.ac.uk/research/groups/hase/javahase/app-list.html>
- Israel Koren at U Massachusetts Amhurst:
 - <http://www.ecs.umass.edu/ece/koren/architecture/Tomasulo/AppletTomasulo.html>
 - <http://www.ecs.umass.edu/ece/koren/architecture/>

Processor performance

- SPEC benchmarks – see <http://www.spec.org/>
 - CPU benchmarks: <http://www.spec.org/cpu2000/results/cpu2000.html>
 - HPC benchmarks: <http://www.spec.org/hpc2002/results/hpc2002.html>
- Ace's hardware SPEC summary:
 - <http://www.aceshardware.com/SPECmine/top.jsp>

Other simulators:

- Liberty: <http://liberty.cs.princeton.edu/>
- MicroLib: <http://microlib.org/>

360/91 design choices...

▶ Speculation:

- ▶ “Rather than wait for a valid CC, fetches are initiated for two instruction double-words as a hedge against a successful branch. Following this, it is assumed that the branch will fail, and a “conditional mode” is established. In conditional mode, shown in Fig. 8, instructions are decoded and conditionally forwarded to the execution units, and concomitant operand fetches are initiated. The execution units are inhibited from completing conditional instructions. When a valid condition code appears, the appropriate branching action is detected and activates or cancels the conditional instructions.”

▶ Prediction:

- ▶ [after mispredict] “the role of conditional mode is reversed, i.e., when the conditional branch is next encountered, it will be assumed that the branch will be taken. The conditionally issued instructions are from the target path rather than from the nbranch path as is the case when not in loop mode. A cancel requires recovery from the branch guess.”

▶ Right:

- ▶ Organizationally, primary emphasis is placed on (1) alleviating the disparity between storage time and circuit speed, and (2) the development of high speed floating-point arithmetic algorithms.

▶ Wrong:

- ▶ “The complications of conditional mode, coupled with the fact that it is primarily aimed at circumventing storage access delays, indicate that a careful re-examination of its usefulness will be called for as the access time decreases.”

Tomasulo Algorithm and Branch Prediction

- ▶ **360/91 predicted branches, but lacked full speculation:**
 - ▶ Instructions along predicted branch path can complete
 - ▶ But results cannot be forwarded until branch outcome resolved
- ▶ **Speculation with Reorder Buffer allows execution past branch, and then discard if branch fails**
 - ▶ The key difference is that speculative instructions can pass values to each other
 - ▶ just need to hold instructions in buffer until branch can commit

Case for Branch Prediction when Issue N instructions per clock cycle

1. Branches will arrive up to n times faster in an n -issue processor
2. Amdahl's Law \Rightarrow relative impact of the control stalls will be larger with the lower potential CPI in an n -issue processor

7 Branch Prediction Schemes

1. **1-bit Branch-Prediction Buffer**
2. **2-bit Branch-Prediction Buffer**
3. **Correlating Branch Prediction Buffer**
4. **Tournament Branch Predictor**
5. **Branch Target Buffer**
6. **Integrated Instruction Fetch Units**
7. **Return Address Predictors**

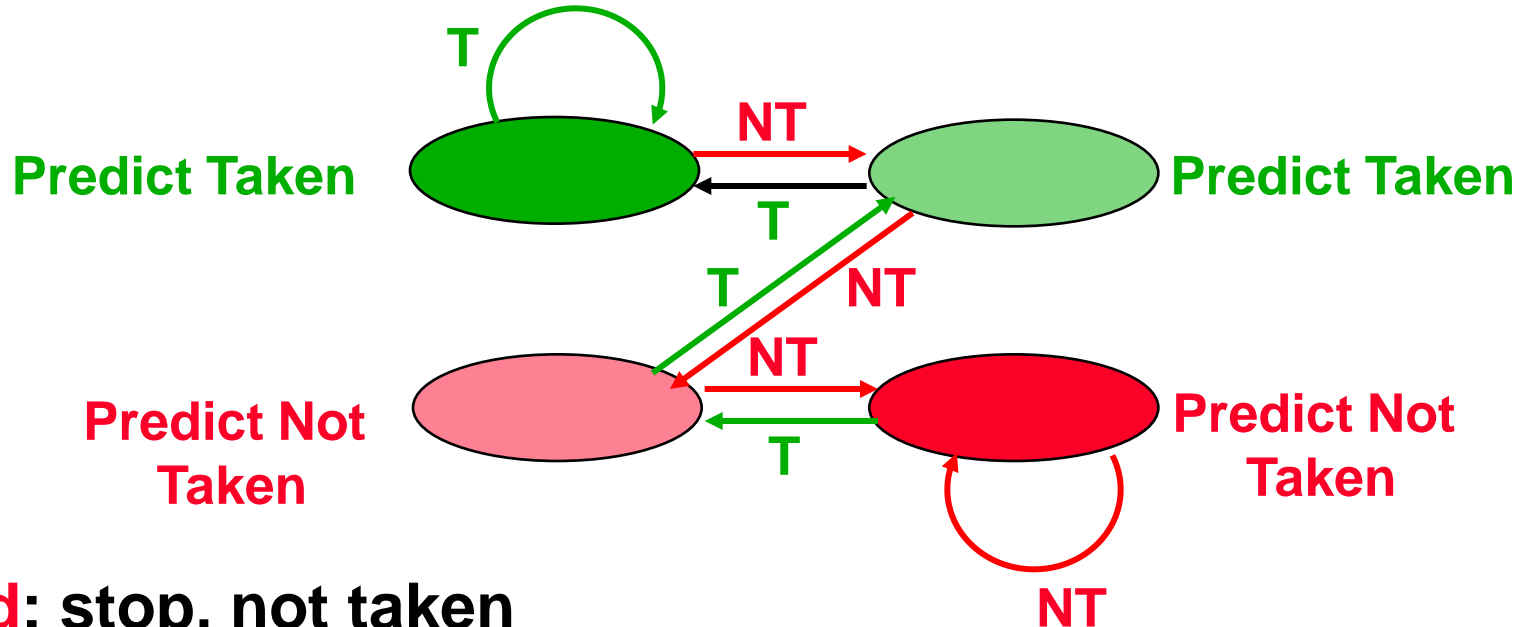
Dynamic Branch Prediction

- ▶ Performance = $f(\text{accuracy, cost of misprediction})$
- ▶ 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 (saves HW, but may not be right branch)
- ▶ Problem: in a loop, 1-bit BHT will cause 2 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
 - ▶ Only 80% accuracy even if loop 90% of the time

Dynamic Branch Prediction

(Jim Smith, 1981)

- Solution: 2-bit scheme where change prediction only if get misprediction *twice*: (Figure 3.7, p. 198)

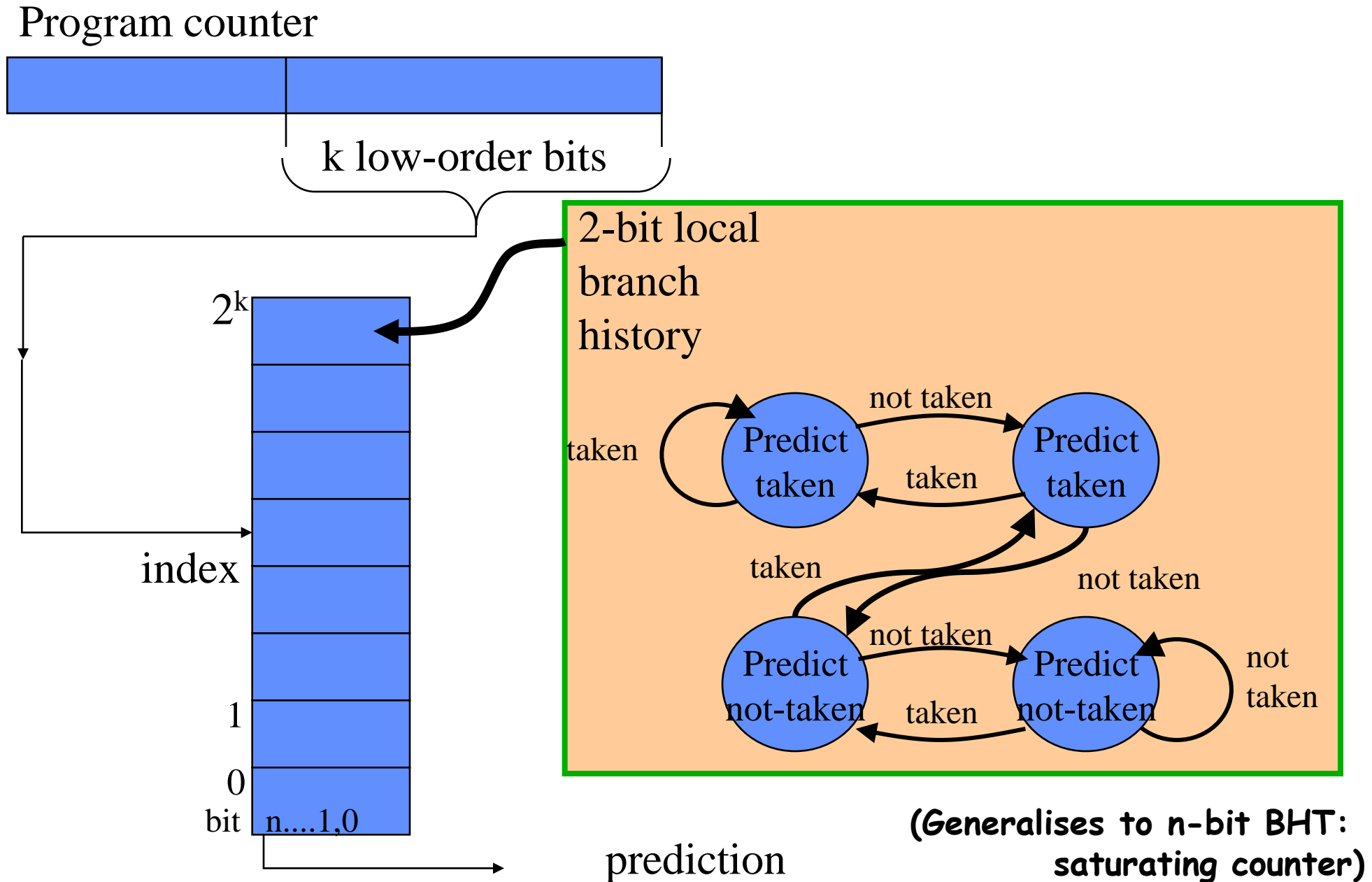


- **Red:** stop, not taken

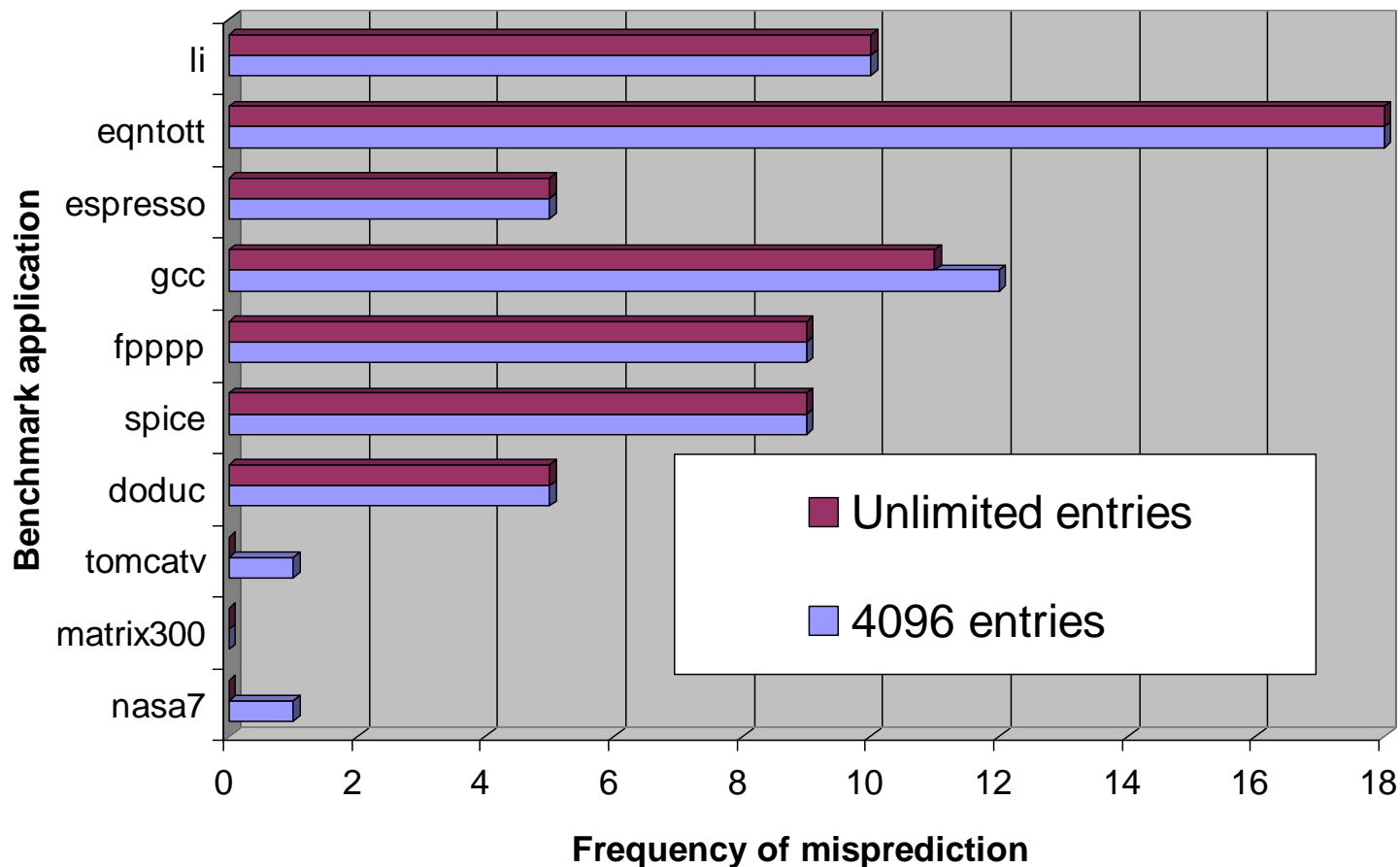
- **Green:** go, taken

- Adds *hysteresis* to decision making process

The 2-bit branch history table (BHT)



Prediction accuracy of an 4096-entry two-bit prediction buffer versus an infinite buffer for the SPEC89 benchmarks (H&P Fig 4.15)



**n-bit BHT
- how well
does it
work?**

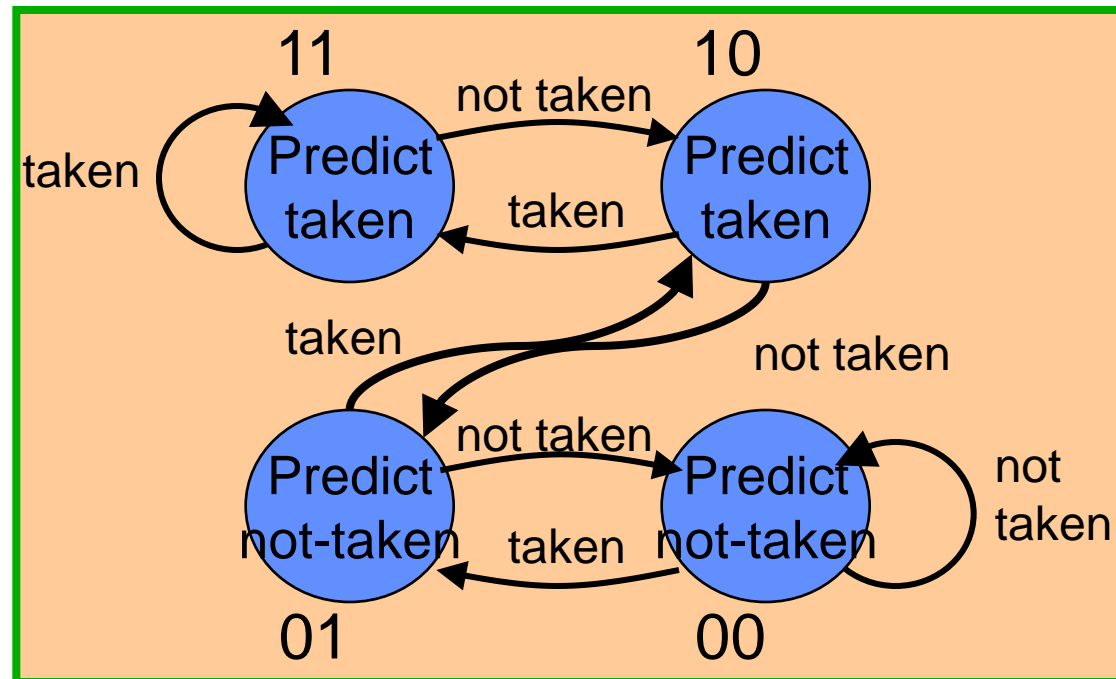
- 2-bit predictor often very good, sometimes awful
- Little evidence that BHT capacity is an issue
- 1-bit is usually worse, 3-bit is not usefully better

N-bit BHT - why does it work so well?

- ▶ n-bit BHT predictor essentially based on a saturating counter: taken increments, not-taken decrements
- ▶ predict taken if most significant bit is set

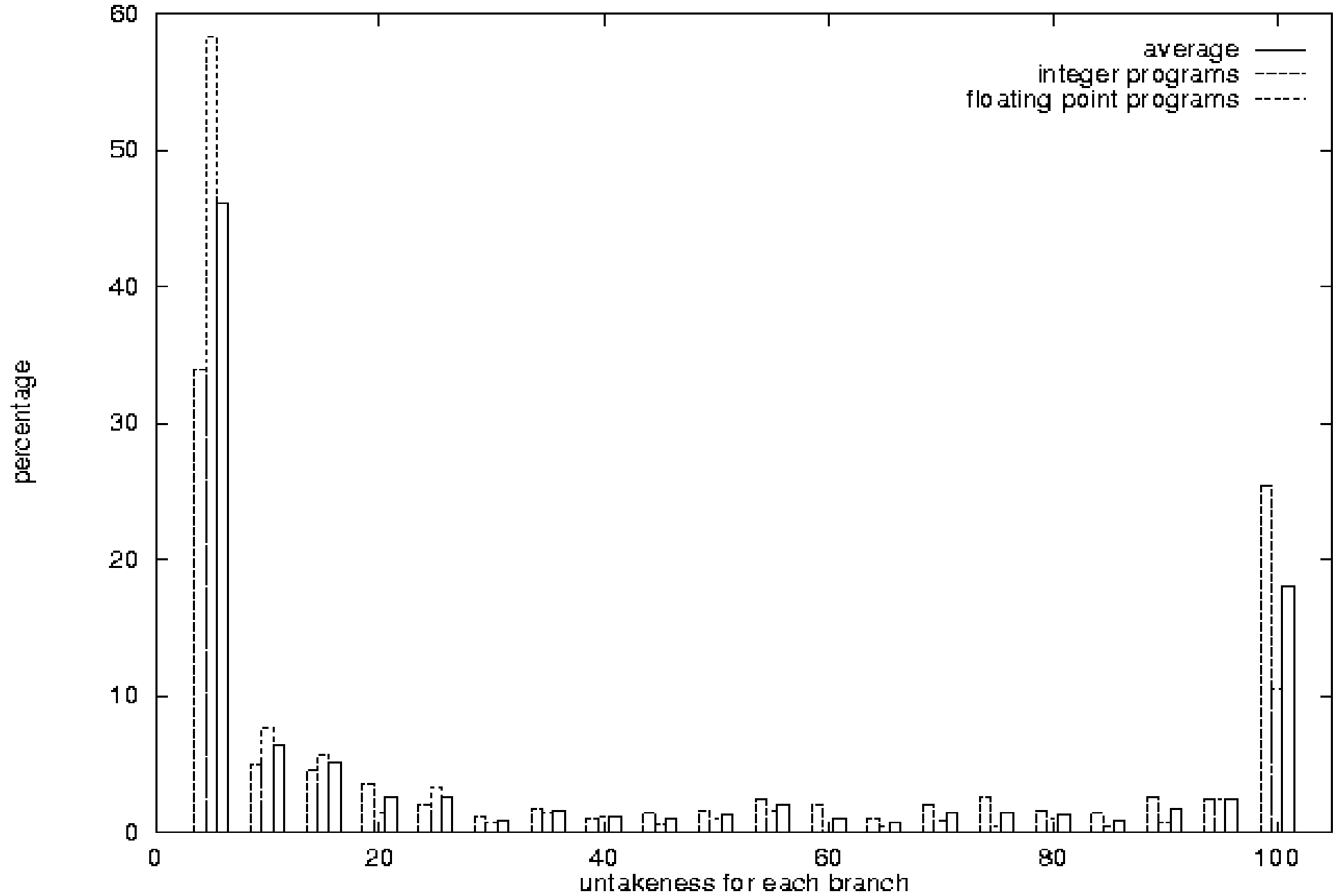
▶ Most branches are highly ***biased***: either almost-always taken, or almost-always not-taken

▶ Works badly for branches which aren't



Often called the “bimodal” predictor

Bias



Is local history all there is to it?

▶ The bimodal predictor uses the BHT to record “local history” - the prediction information used to predict a particular branch is determined only by its memory address

▶ Consider the following sequence:

© It is very likely that condition C2 is correlated with C1 - and that C3 is correlated with C1 and C2

© How can we use this observation?

```
if (C1) then
    S1;
endif
if (C2) then
    S2;
endif
if (C3) then
    S3;
endif
```

Global history

- ▶ Definition: **Global history**. The taken - not-taken history for all previously-executed branches.
- ▶ Idea: use global history to improve branch prediction
- ▶ Compromise: use m most recently-executed branches
- ▶ Implementation: keep an m -bit Branch History Register (BHR) - a shift register recording taken - not-taken direction of the last m branches
- ▶ Question: How to combine local information with global information?

Branch history register

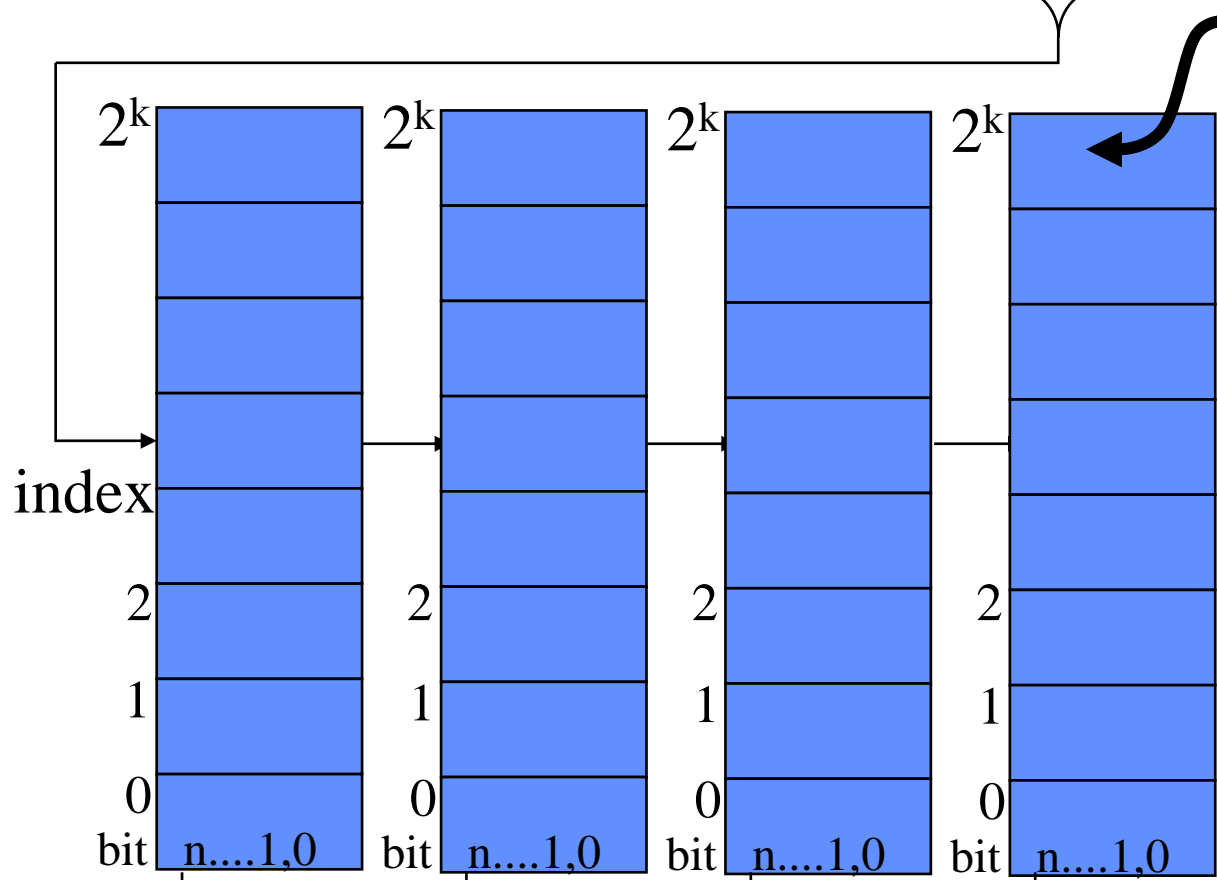
Program counter

m bits

k low-order bits

This is an (m, n) "gselect" correlating predictor:

- ➔ m global bits record behaviour of last m branches
- ➔ These m bits are used to select which of the 2^m n-bit BHTs to use



Popular choice is m=2, n=2, so four tables each of 2×2^k bits

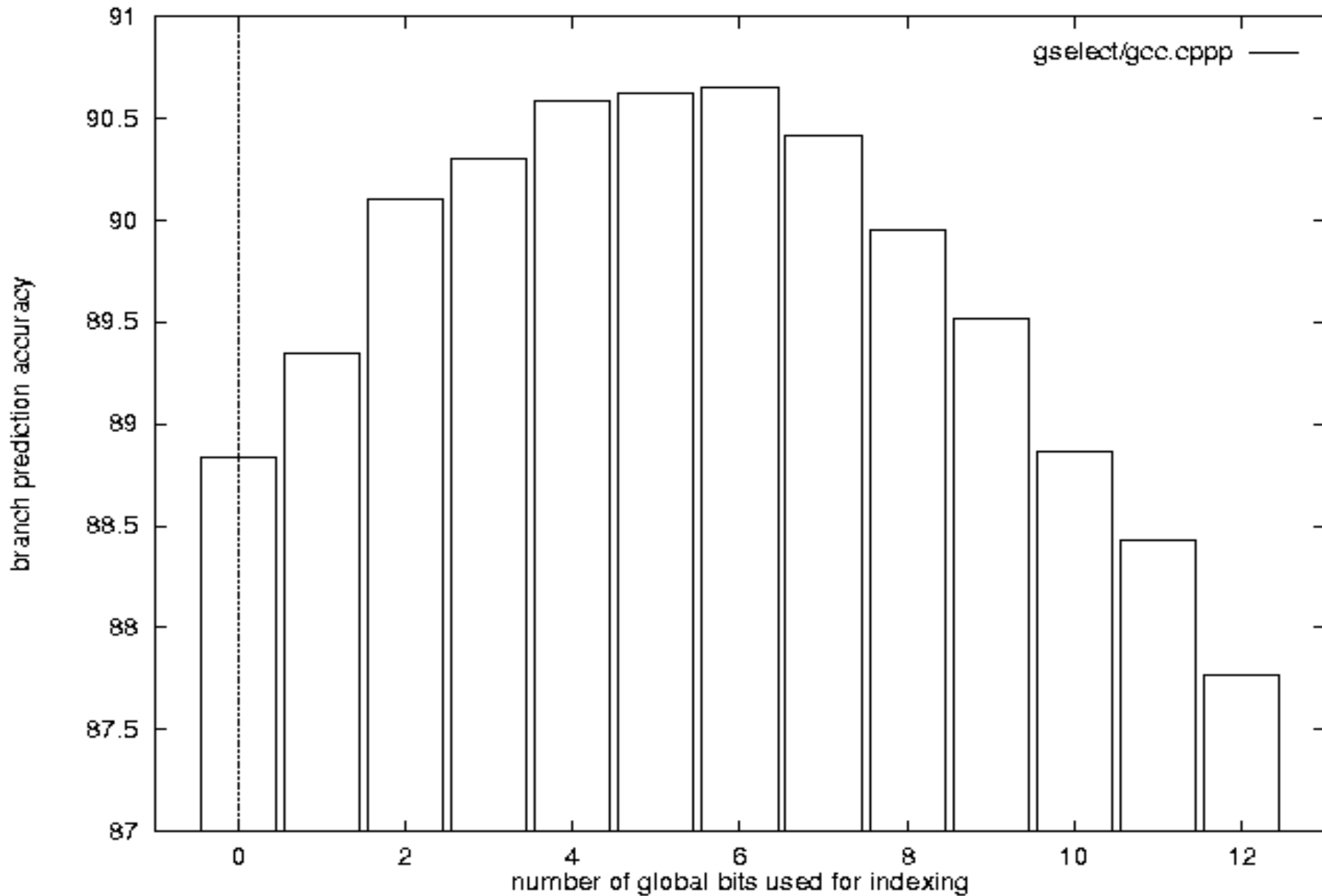
Select



prediction

2^m n-bit BHTs

How many bits of branch history should be used?



 (2,2) is good, (4,2) is better, (10,2) is worse

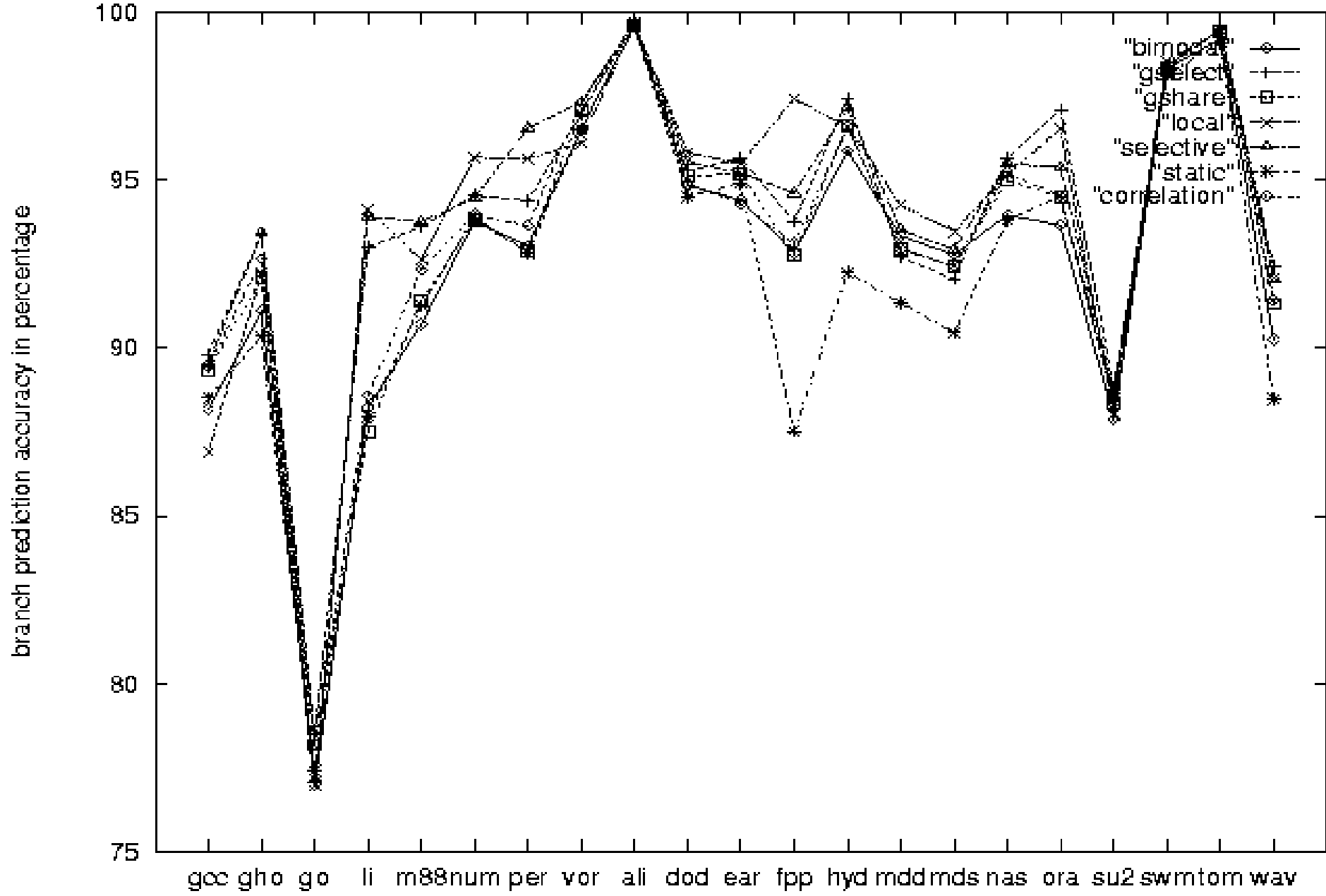
Variations

▶ There are many variations on the idea:

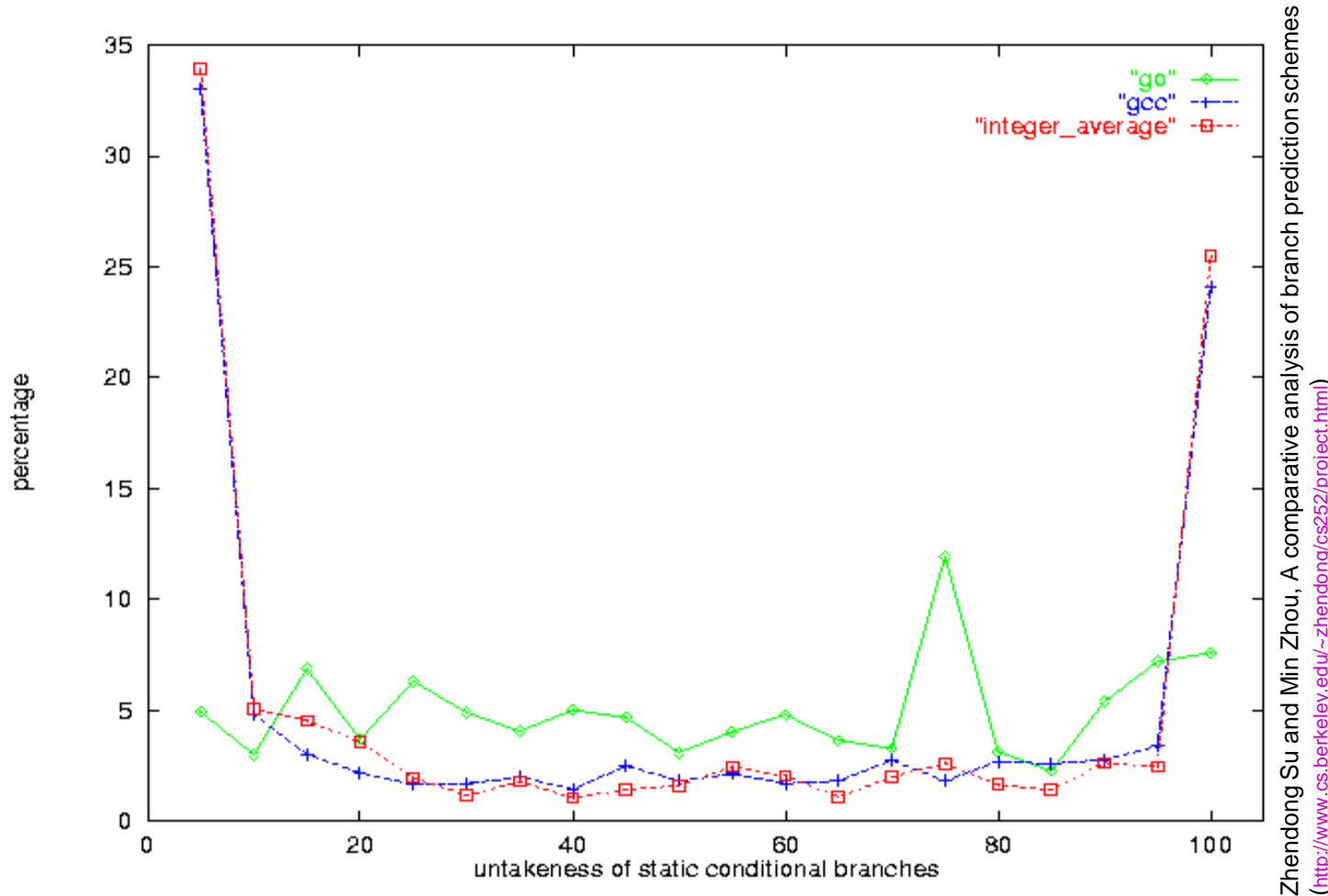
- ▶ *gselect*: many combinations of n and m
- ▶ *global*: use *only* the global history to index the BHT - ignore the PC of the branch being predicted (an extreme (n,m) *gselect* scheme)
- ▶ *gshare*: arrange bimodal predictors in single BHT, but construct its index by XORing low-order PC address bits with global branch history shift register - claimed to reduce conflicts
- ▶ *Per-address Two-level Adaptive using Per-address pattern history (PAP)*: for each branch, keep a k -bit shift register recording its history, and use this to index a BHT *for this branch* (see Yeh and Patt, 1992)

▶ Each suits some programs well but not all

Horses for courses



Extreme example - “go”

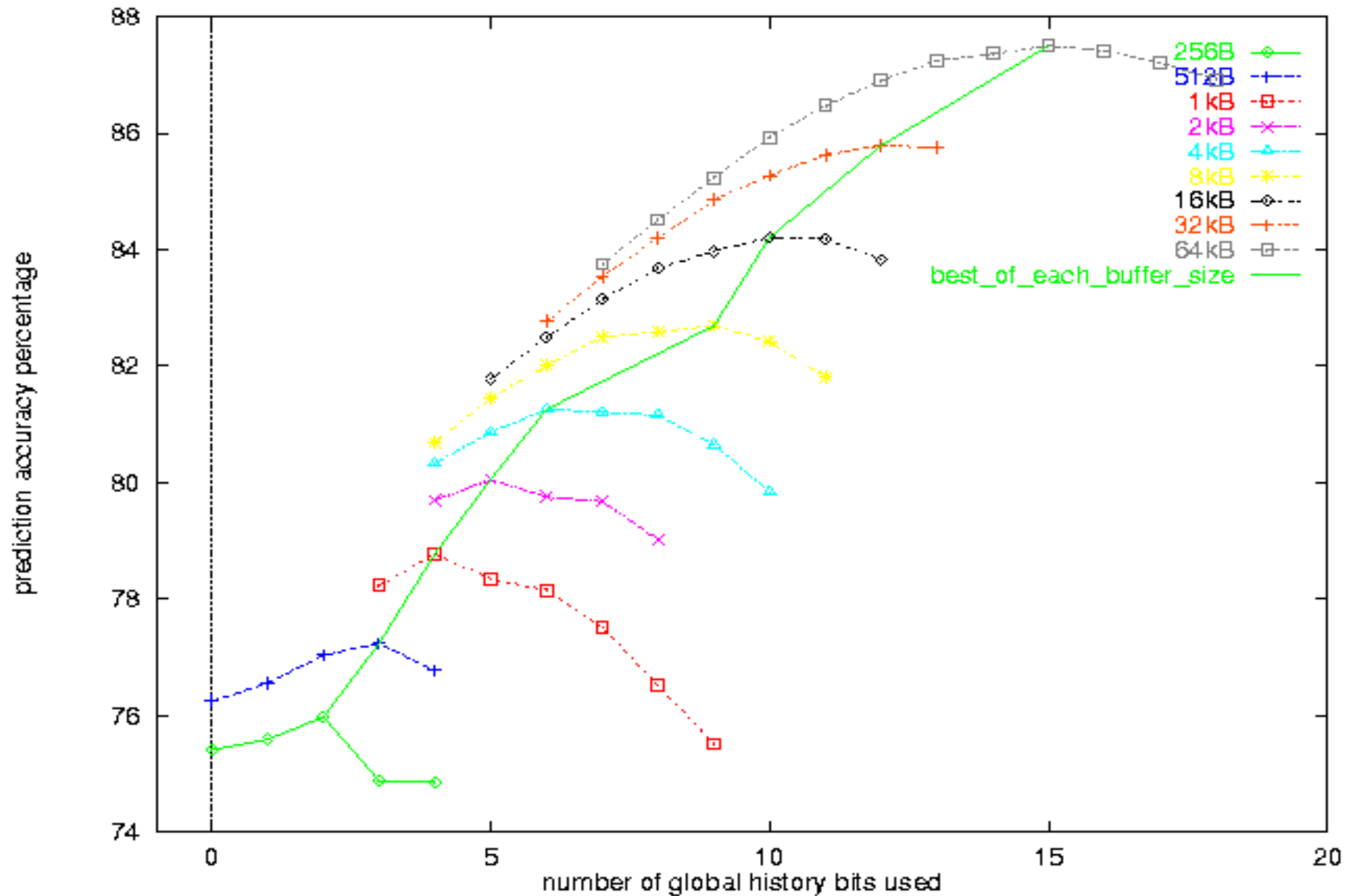


Zhendong Su and Min Zhou, A comparative analysis of branch prediction schemes (<http://www.cs.berkeley.edu/~zhendong/cs252/project.htm>)

“go” is a SPEC95 benchmark code with highly-dynamic, highly-correlated branch behaviour

- The bias of “go”’s branches is more-or-less evenly spread between 0% taken and 100% taken
- All known predictors do badly

Some dynamic applications have highly-correlated branches



For “go”, optimum BHR size (m) is much larger

Re-evaluating Correlation

- Several of the SPEC benchmarks have less than a dozen branches responsible for 90% of taken branches:

program	branch %	static	# = 90%
compress	14%	236	13
<u>eqntott</u>	<u>25%</u>	<u>494</u>	<u>5</u>
gcc	15%	9531	2020
mpeg	10%	5598	532
real gcc	13%	17361	3214

- Real programs + OS more like gcc
- Small benefits beyond benchmarks for correlation? problems with branch aliases?

Predicated Execution

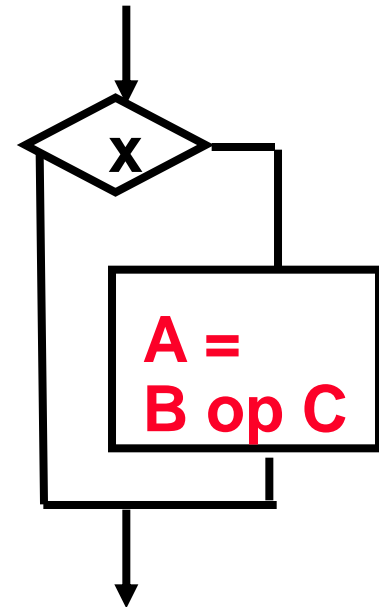
➤ Avoid branch prediction by turning branches into conditionally executed instructions:

if (x) then A = B op C else NOP

- If false, then neither store result nor cause exception
- Expanded ISA of Alpha, MIPS, PowerPC, SPARC have conditional move; PA-RISC can annul any following instr.
- IA-64: 64 1-bit condition fields selected so conditional execution of any instruction
- This transformation is called **“if-conversion”**

➤ Drawbacks to conditional instructions

- Still takes a clock even if “annulled”
- Stall if condition evaluated late
- Complex conditions reduce effectiveness; condition becomes known late in pipeline



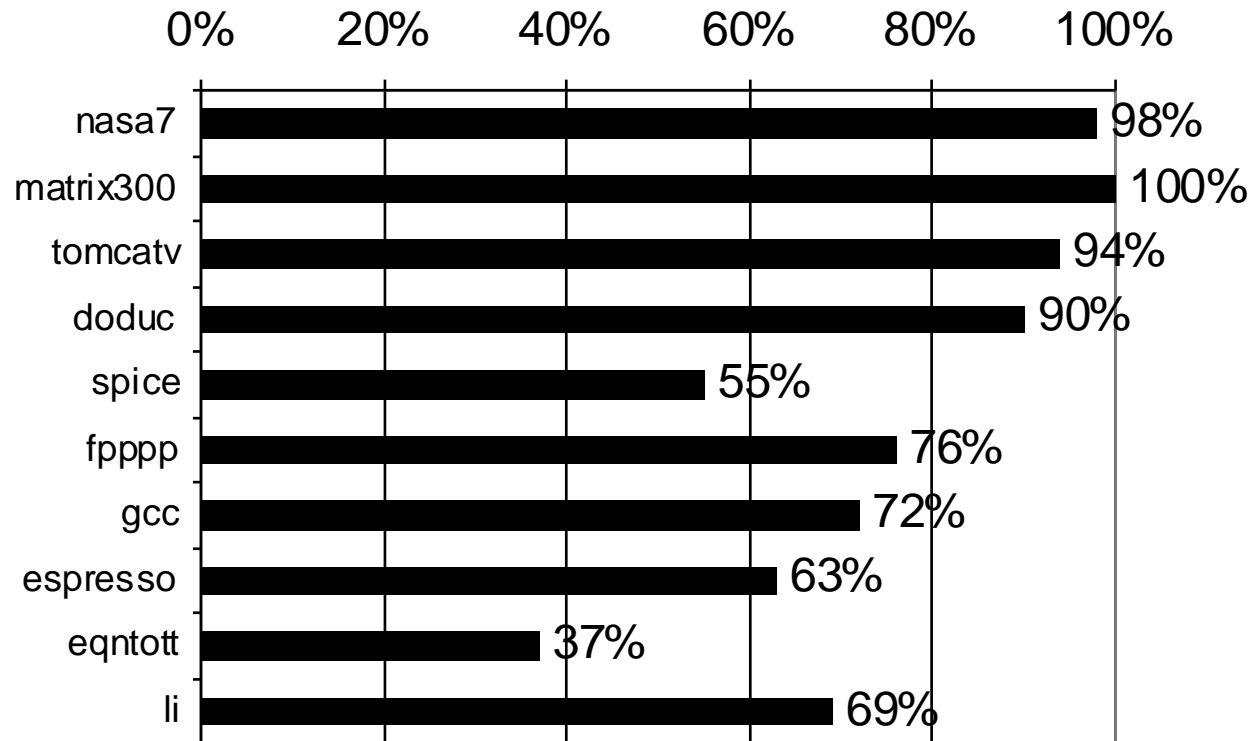
Tournament Predictors

- ▶ Motivation for correlating branch predictors is 2-bit predictor failed on important branches; by adding global information, performance improved
- ▶ Tournament predictors: use 2 predictors, 1 based on global information and 1 based on local information, and combine with a selector
- ▶ Hopes to select right predictor for right branch

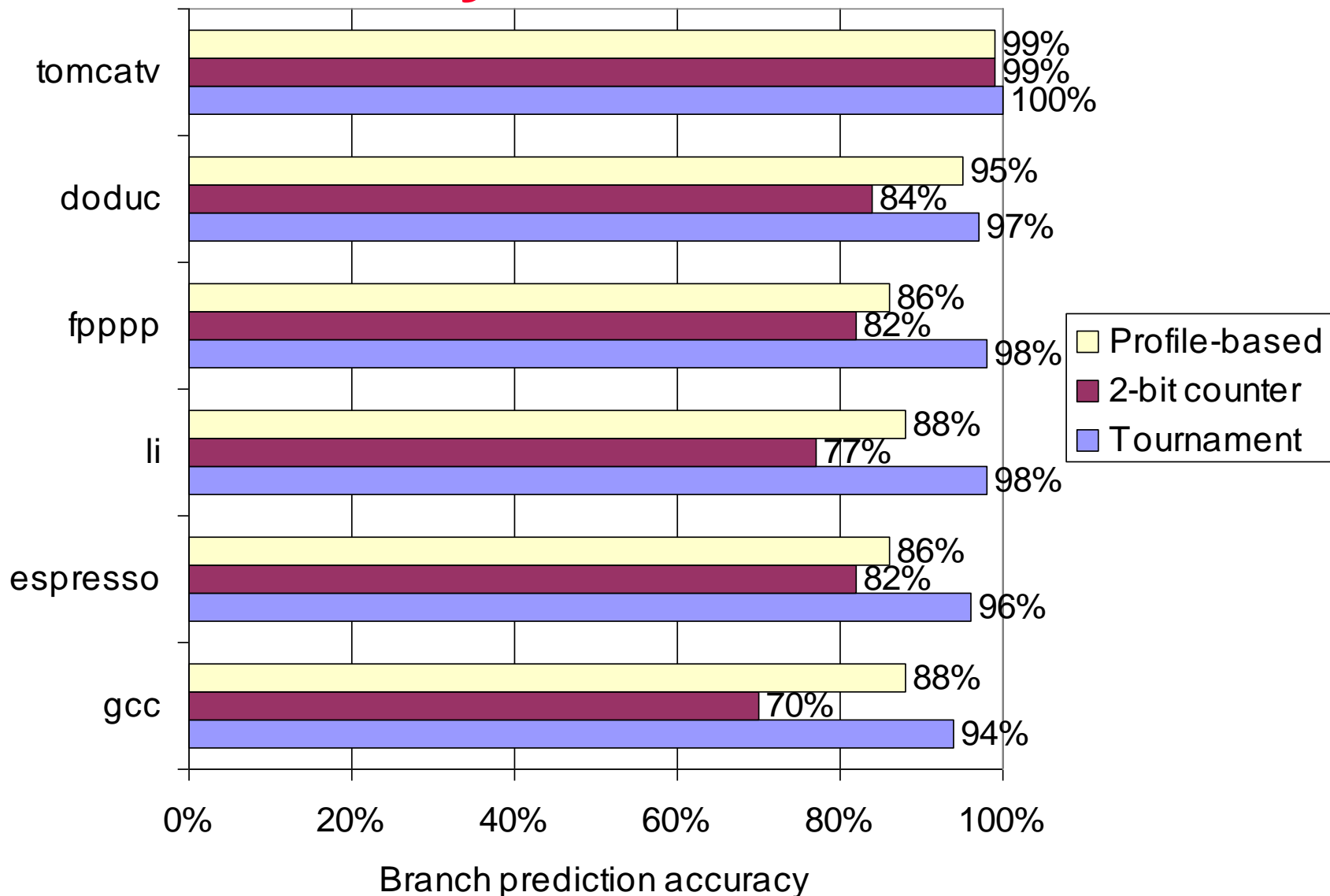
Tournament Predictor in Alpha 21264

- ▶ 4K 2-bit counters to choose from among a global predictor and a local predictor
- ▶ **Global predictor** also has 4K entries and is indexed by the history of the last 12 branches; each entry in the global predictor is a standard 2-bit predictor
 - ▶ 12-bit pattern: ith bit 0 => ith prior branch not taken;
ith bit 1 => ith prior branch taken;
- ▶ **Local predictor** consists of a 2-level predictor:
 - ▶ **Top level** a local history table consisting of 1024 10-bit entries; each 10-bit entry corresponds to the most recent 10 branch outcomes for the entry. 10-bit history allows patterns 10 branches to be discovered and predicted.
 - ▶ **Next level** Selected entry from the local history table is used to index a table of 1K entries consisting a 3-bit saturating counters, which provide the local prediction
- ▶ Total size: $4K \cdot 2 + 4K \cdot 2 + 1K \cdot 10 + 1K \cdot 3 = 29K$ bits!
(~180,000 transistors)

% of predictions from local predictor in Tournament Prediction Scheme

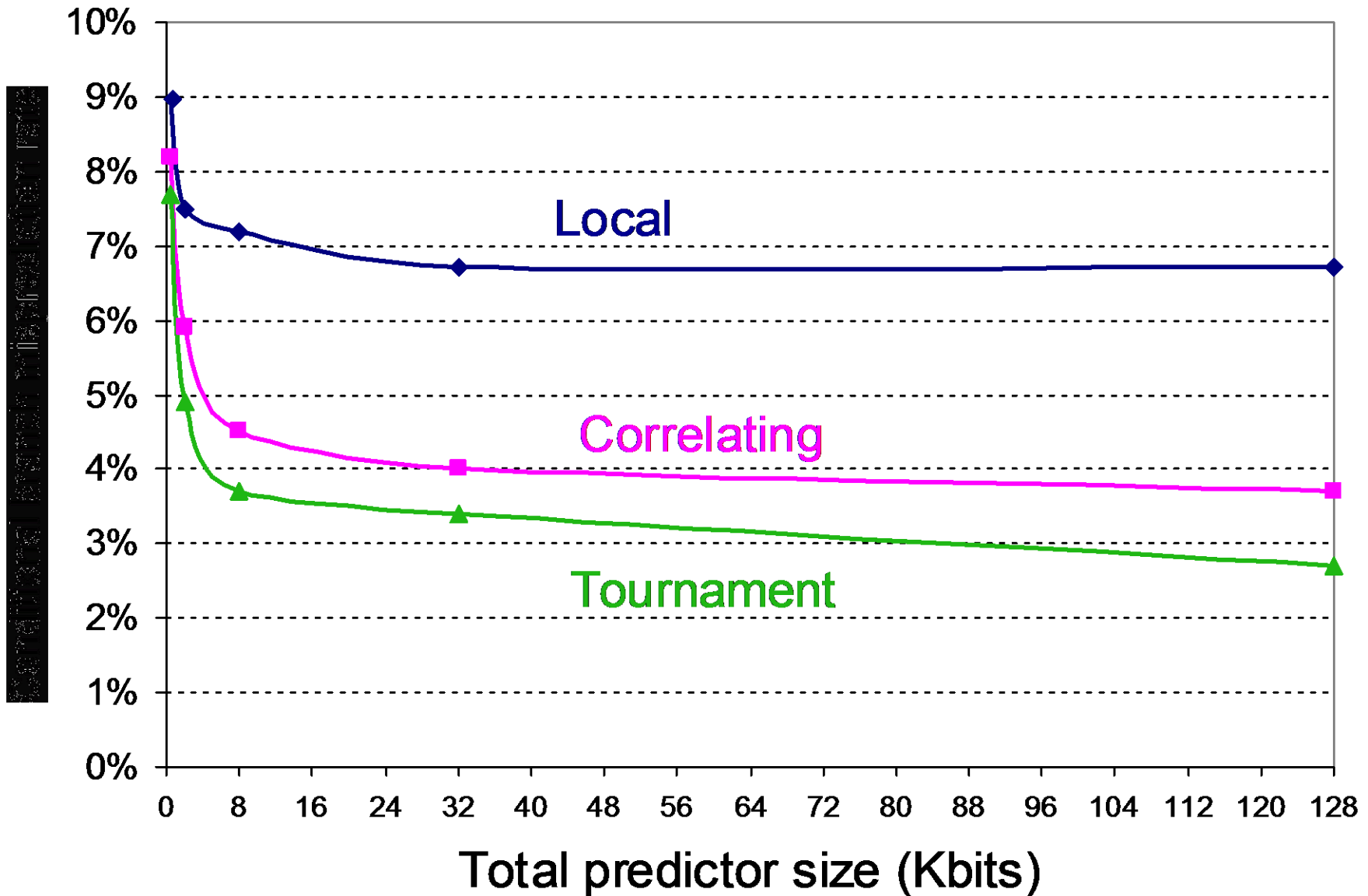


Accuracy of Branch Prediction



 **Profile: branch profile from last execution
(static in that it is encoded in instruction, but profile)**

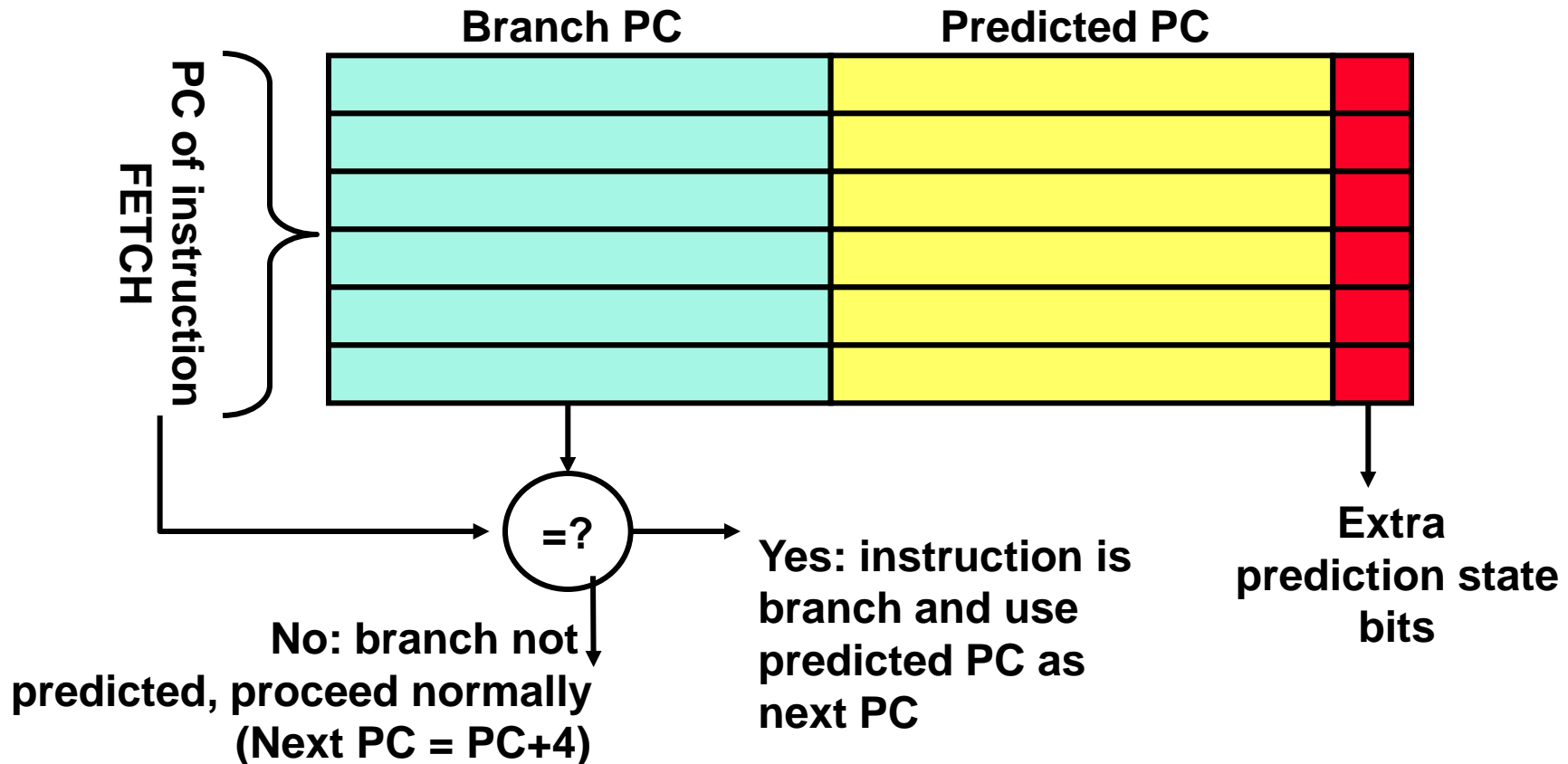
Accuracy v. Size (SPEC89)



Need Address at Same Time as Prediction

Branch Target Buffer (BTB): Address of branch index to get prediction AND branch address (if taken)

- Note: must check for branch match now, since can't use wrong branch address (Figure 3.19, p. 262)



Special Case Return Addresses

- ▶ Register Indirect branch hard to predict address
- ▶ SPEC89 85% such branches for procedure return
- ▶ Since stack discipline for procedures, save return address in small buffer that acts like a stack: 8 to 16 entries has small miss rate

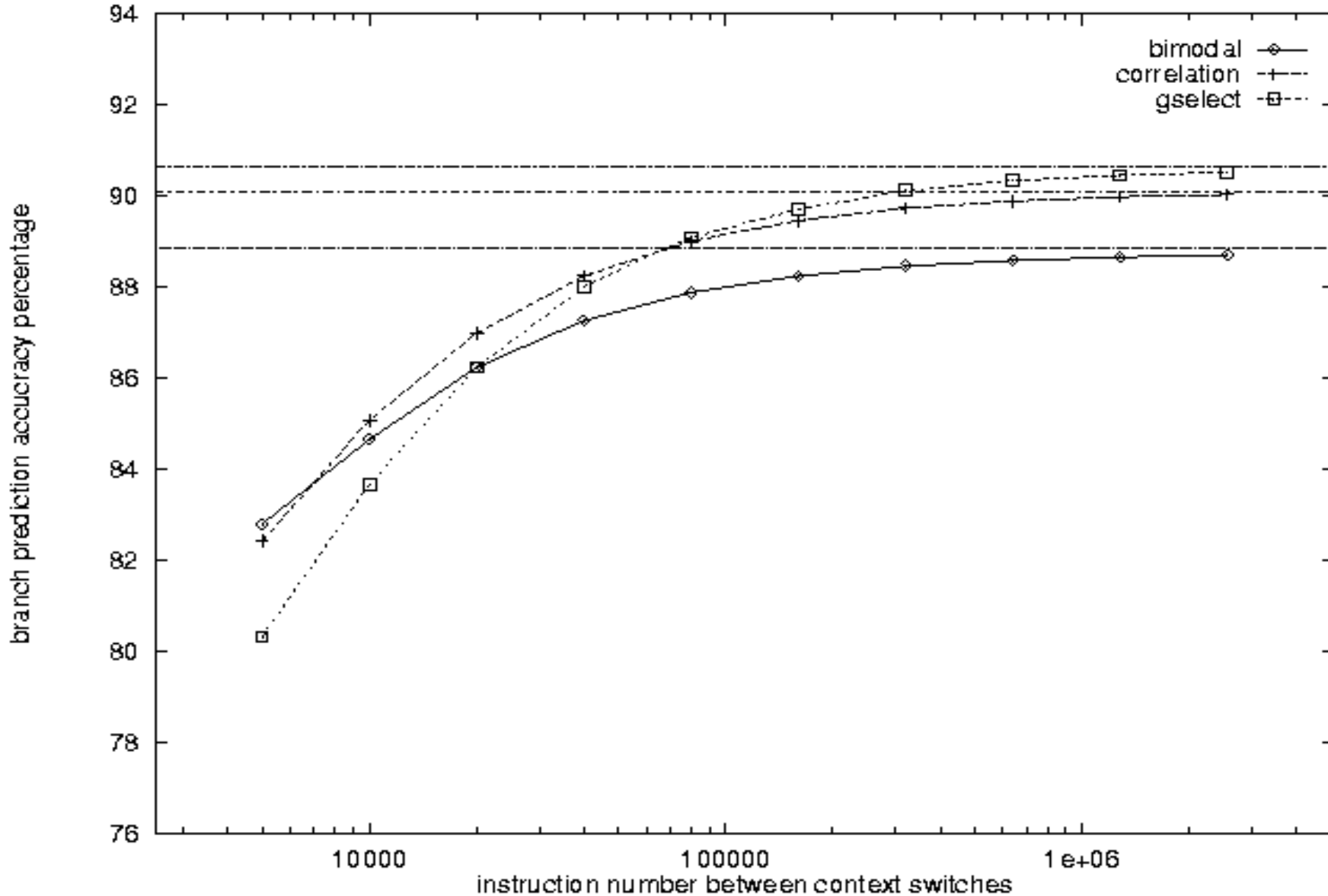
Pitfall: Sometimes bigger and dumber is better

- ▶ **21264** uses tournament predictor (29 Kbits)
- ▶ Earlier **21164** uses a simple 2-bit predictor with 2K entries (or a total of 4 Kbits)
- ▶ **SPEC95 benchmarks, 21264 outperforms**
 - ▶ **21264** avg. 11.5 mispredictions per 1000 instructions
 - ▶ **21164** avg. 16.5 mispredictions per 1000 instructions
- ▶ **Reversed for transaction processing (TP) !**
 - ▶ **21264** avg. 17 mispredictions per 1000 instructions
 - ▶ **21164** avg. 15 mispredictions per 1000 instructions
- ▶ **TP code much larger & 21164 hold 2X branch predictions based on local behavior (2K vs. 1K local predictor in the 21264)**

Warm-up effects and context-switching

- ▶ In real life, applications are interrupted and some other program runs for a while (if only the OS)
- ▶ This means the branch prediction is regularly trashed
- ▶ Simple predictors re-learn fast
 - ▶ in 2-bit bimodal predictor, all executions of given branch update same 2 bits
- ▶ Sophisticated predictors re-learn more slowly
 - ▶ for example, in (2,2) gselect predictor, prediction updates are spread across 4 BHTs
- ▶ *Selective* predictor may choose fast learner predictor until better predictor warms up

Warm-up..



Best predictor takes 20,000 instructions to overtake bimodal

Dynamic Branch Prediction Summary

- ▶ Prediction becoming important part of scalar execution
- ▶ Branch History Table: 2 bits for loop accuracy
 - ▶ Saturating counter (bimodal) scheme handles highly-biased branches well
 - ▶ Some applications have highly dynamic branches
- ▶ Correlation: Recently executed branches correlated with next branch.
 - ▶ Either different branches
 - ▶ Or different executions of same branches
- ▶ Tournament Predictor: more resources to competitive solutions and pick between them
- ▶ Branch Target Buffer: include branch address & prediction
- ▶ Predicated Execution can reduce number of branches, number of mispredicted branches
- ▶ Return address stack for prediction of indirect jump

Branch prediction resources

- ▶ **Design tradeoffs for the Alpha EV8 Conditional Branch Predictor (André Seznec, Stephen Felix, Venkata Krishnan, Yiannakis Sazeides)**
 - ▶ SMT: 4 threads, wide-issue superscalar processor, 8-way issue, 512 registers (cancelled June 2001 when Alpha dropped)
 - ▶ Paper: <http://citeseer.ist.psu.edu/seznec02design.html>
 - ▶ Talk: <http://ce.et.tudelft.nl/cecoll/slides/PresDelft0803.ppt>
- ▶ **Branch prediction in the Pentium family (Agner Fog)**
 - ▶ Reverse engineering Pentium branch predictors using direct access to BTB
 - ▶ <http://www.x86.org/articles/branch/branchprediction.htm>
- ▶ **Championship Branch Prediction Competition (CBP-1), organised by the Journal of Instruction-level Parallelism**
 - ▶ <http://www.jilp.org/cbp/>

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 (TBD)
 - ▶ Intel Architecture-64 (IA-64) 64-bit address
 - Renamed: “Explicitly Parallel Instruction Computer (EPIC)”
 - ▶ Will discuss shortly
- ▶ 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	PipeStages						
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

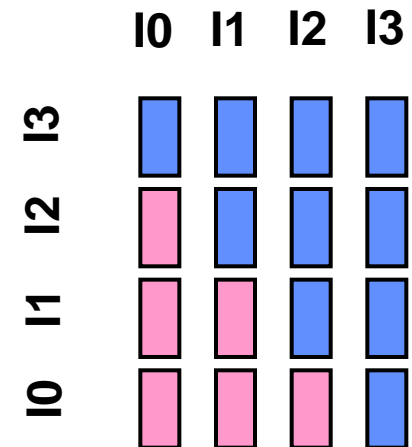
Multiple Issue Issues

▶ **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: $O(n^2-n)$ comparisons

- ▶ 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; (N-issue $\sim O(N^2-N)$ comparisons)
- ▶ 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

- ▶ 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

Dynamic Scheduling in Superscalar

The easy way

- ▶ **How to issue two instructions and keep in-order instruction issue for Tomasulo?**
 - ▶ Assume 1 integer + 1 floating point
 - ▶ 1 Tomasulo control for integer, 1 for floating point
- ▶ **Issue 2X Clock Rate, so that issue remains in order**
- ▶ **Only loads/stores might cause dependency between integer and FP issue:**
 - ▶ Replace load reservation station with a load queue; operands must be read in the order they are fetched
 - ▶ Load checks addresses in Store Queue to avoid RAW violation
 - ▶ Store checks addresses in Load Queue to avoid WAR,WAW

Register renaming, virtual registers versus Reorder Buffers

- ▶ Alternative to Reorder Buffer is a larger virtual set of registers and register renaming
- ▶ **Virtual registers** hold both architecturally visible registers + temporary values
 - ▶ replace functions of reorder buffer and reservation station
- ▶ Renaming process maps names of architectural registers to registers in virtual register set
 - ▶ Changing subset of virtual registers contains architecturally visible registers
- ▶ Simplifies instruction commit: mark register as no longer speculative, free register with old value
- ▶ Adds 40-80 extra registers: Alpha, Pentium,...
 - ▶ Size limits no. instructions in execution (used until commit)

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: aggressive designs may speculate across multiple branches!**

▶ **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?**

- ▶ **Eg vector/SIMD instruction set extensions (SSE, AltiVec etc)**

Limits to ILP

Initial HW Model here; MIPS compilers.

Assumptions for ideal/perfect machine to start:

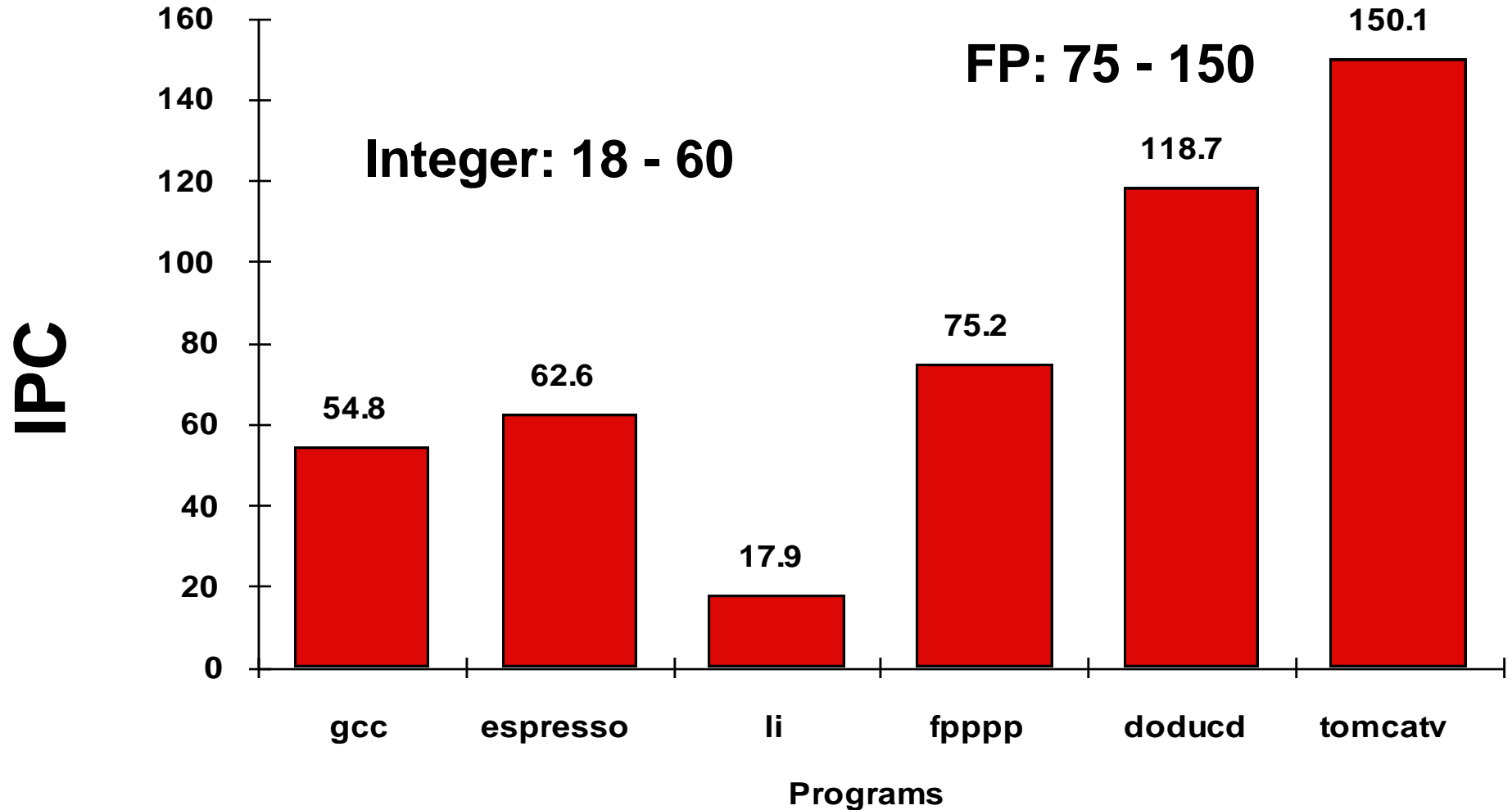
1. **Register renaming** – infinite virtual registers
=> all register WAW & WAR hazards are avoided
2. **Branch prediction** – perfect; no mispredictions
3. **Jump prediction** – all jumps perfectly predicted
2 & 3 => machine with perfect speculation & an unbounded buffer of instructions available
4. **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 *,/);

Upper Limit to ILP: Ideal Machine

(H&P3ed Figure 3.35, page 242)



More Realistic HW: Branch Impact

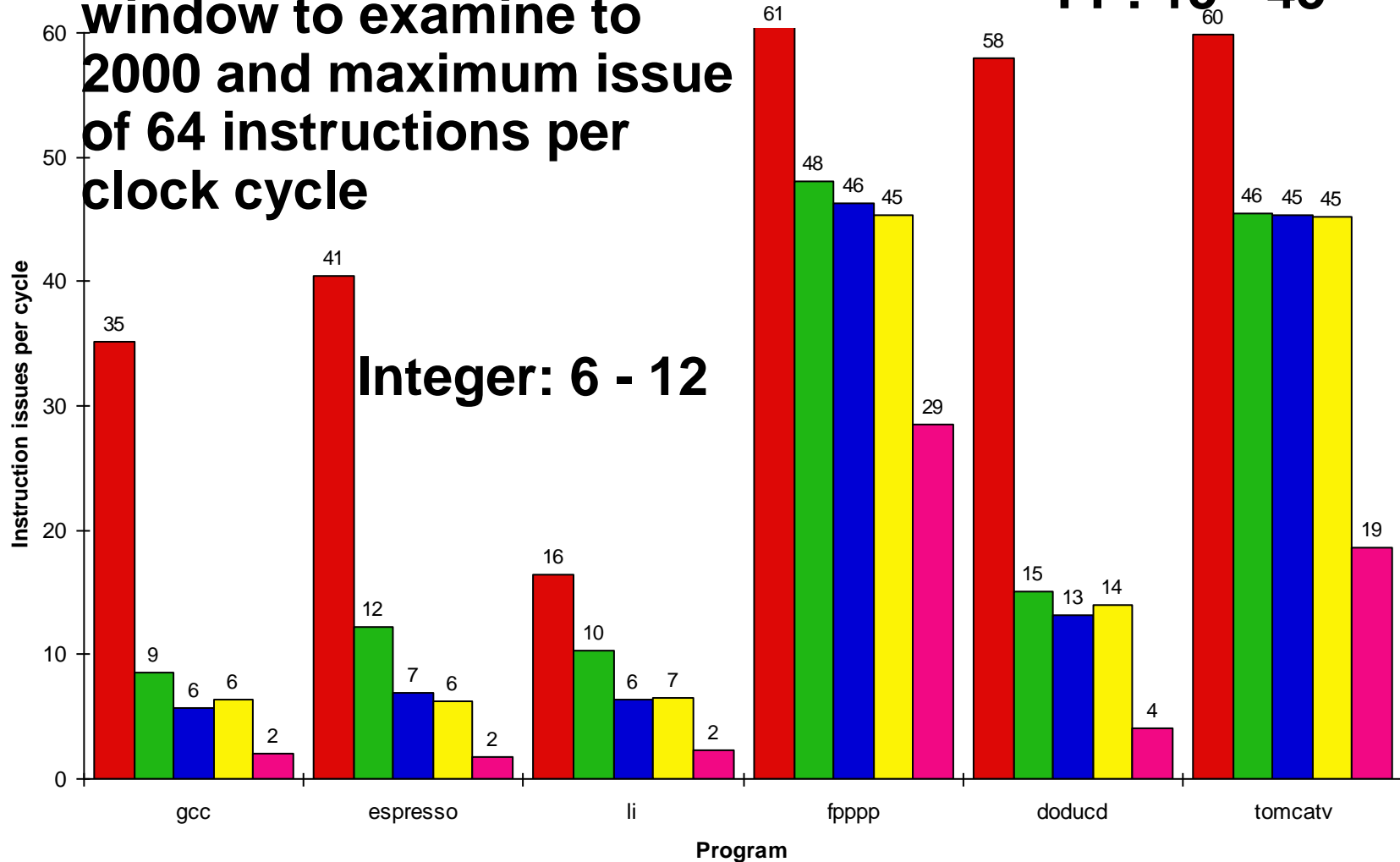
cf H&P3ed Figure 3.39, Page 248

Change from Infinite window to examine to 2000 and maximum issue of 64 instructions per clock cycle

FP: 15 - 45

Integer: 6 - 12

IPC



Perfect

Tournament

BHT (512)

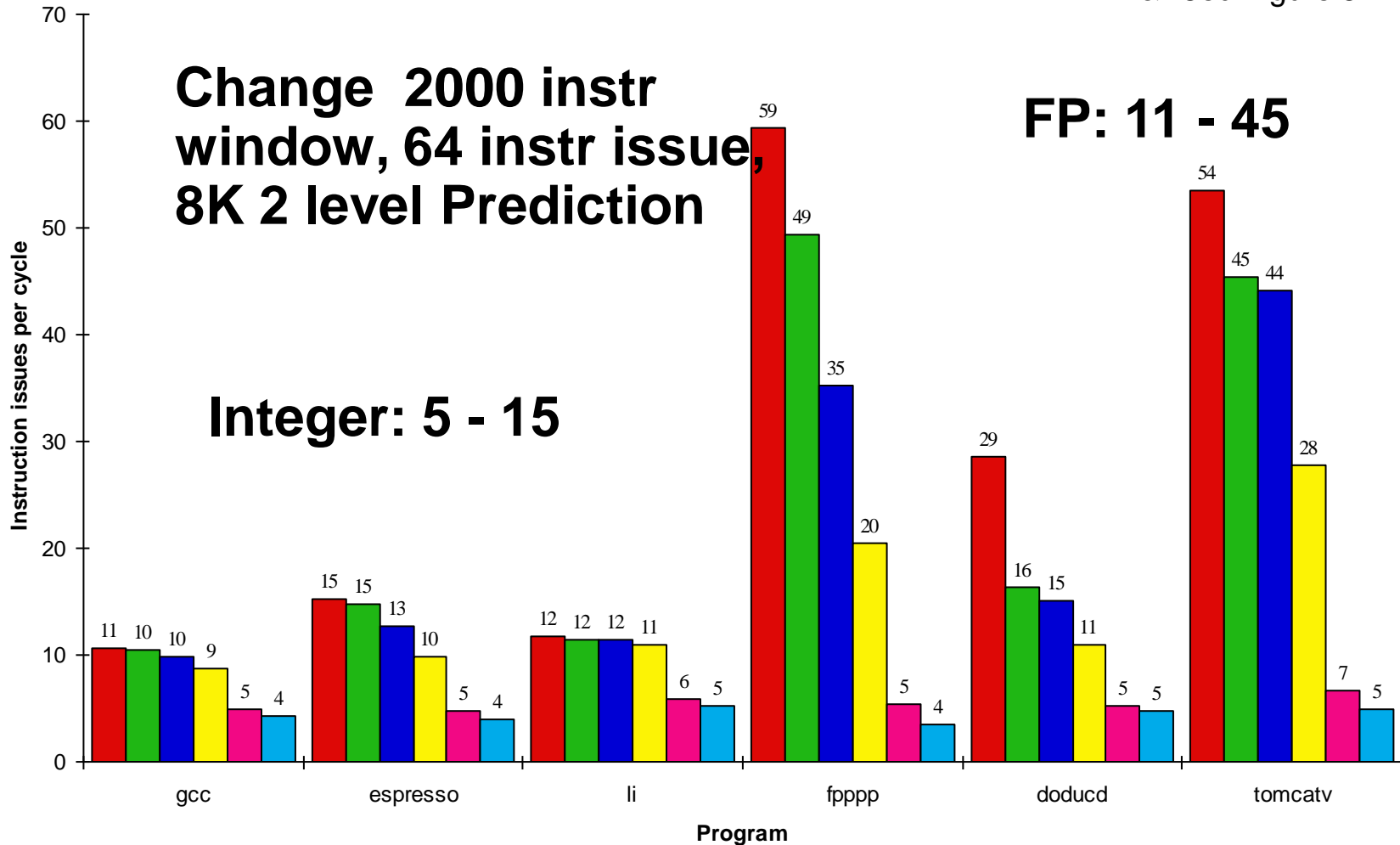
Profile

No prediction

More Realistic HW: Renaming Register Impact

H&P3ed Figure 3.42, Page 251

IPC



Infinite

256

128

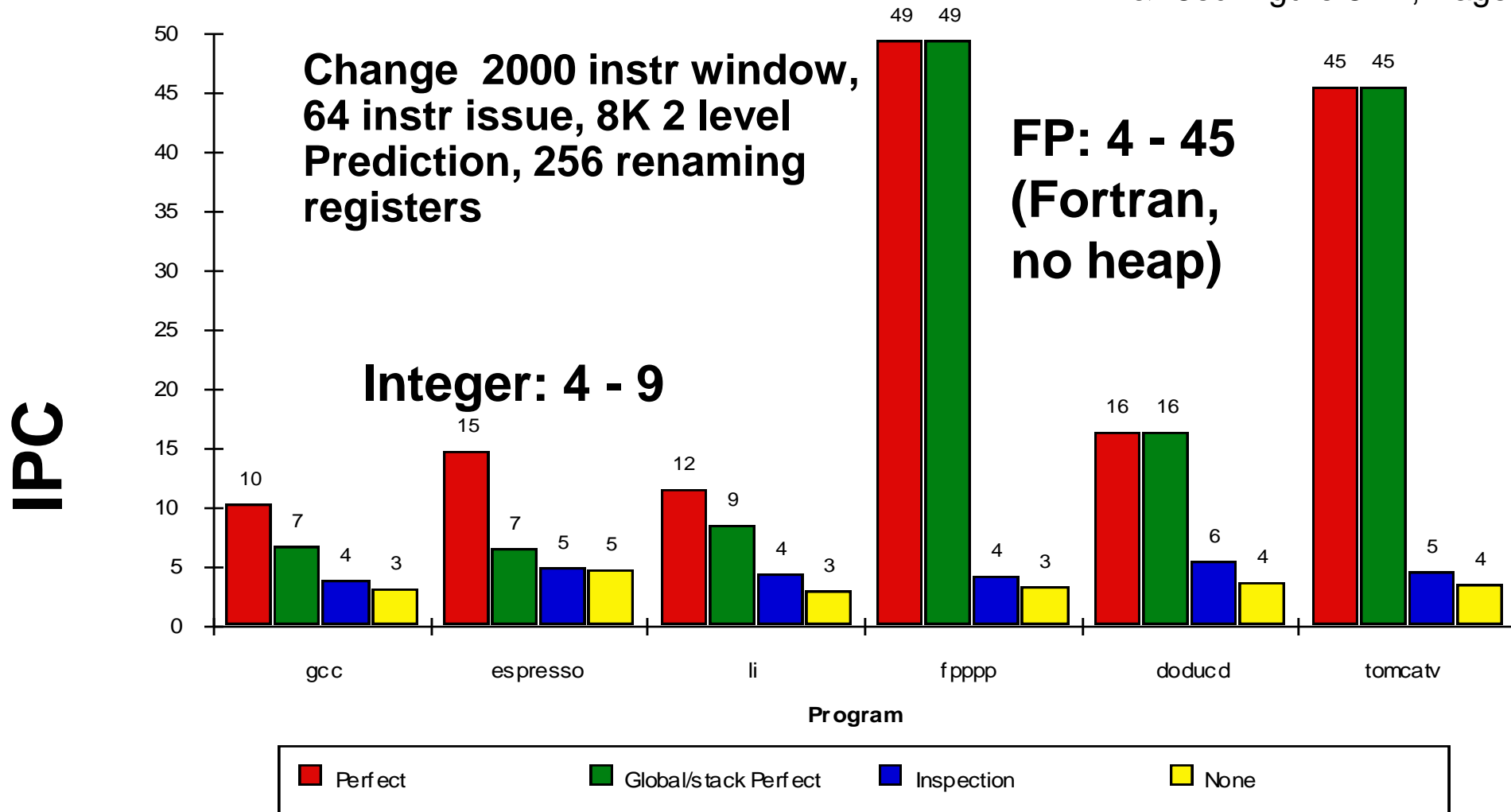
64

32

None

More Realistic HW: Memory Address Alias Impact

H&P3ed Figure 3.44, Page 252



Perfect

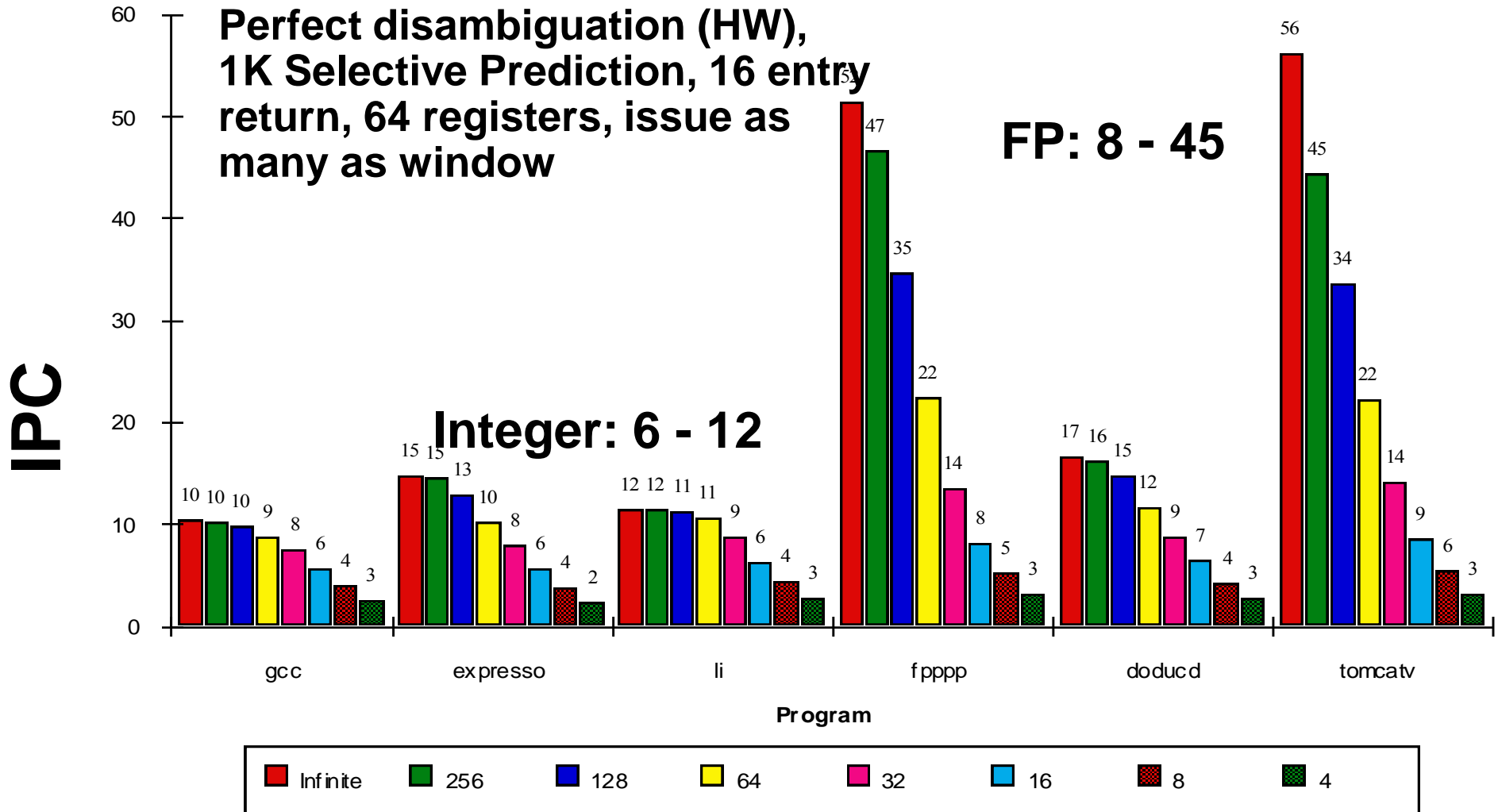
**Global/Stack perf;
heap conflicts**

**Inspe.
Assem.**

None

Realistic HW for '00: Window Impact

(Figure 3.45, Page 309)



Infinite

256

128

64

32

16

8

4

Limits to ILP - resources

- ▶ Limits of Control Flow on Parallelism .
Monica S. Lam, Robert P. Wilson.
19th ISCA, May 1992, pages 19-21.
- ▶ Limits of Instruction-Level Parallelism .
David W. Wall.
DEC-WRL Research Report 93/6, Nov. 1993
- ▶ The Distribution of Instruction-Level and Machine Parallelism and Its Effect on Performance .
Norman P. Jouppi.
IEEE Transactions on Computers, Dec. 1989.

How to Exceed ILP Limits of this study?

- ▶ WAR and WAW hazards through memory: eliminated WAW and WAR hazards through register renaming, but not in memory usage
- ▶ Unnecessary dependences (compiler not unrolling loops so iteration variable dependence)
- ▶ Overcoming the data flow limit: **value prediction**, predicting values and speculating on prediction
 - ▶ **Address value prediction and speculation** predicts addresses and speculates by reordering loads and stores; could provide better aliasing analysis, only need predict if addresses =

Value Locality and Load Value Prediction. Mikko H. Lipasti, Christopher B. Wilkerson, John Paul Shen. Slides by Kundan Nepal:

http://www.lems.brown.edu/~iris/en291s9-04/lectures/kundanvalue_pred.pdf

How to Exceed ILP Limits of this study?

Vector instructions

➡ Next section of this Chapter

Simultaneous Multi-threading

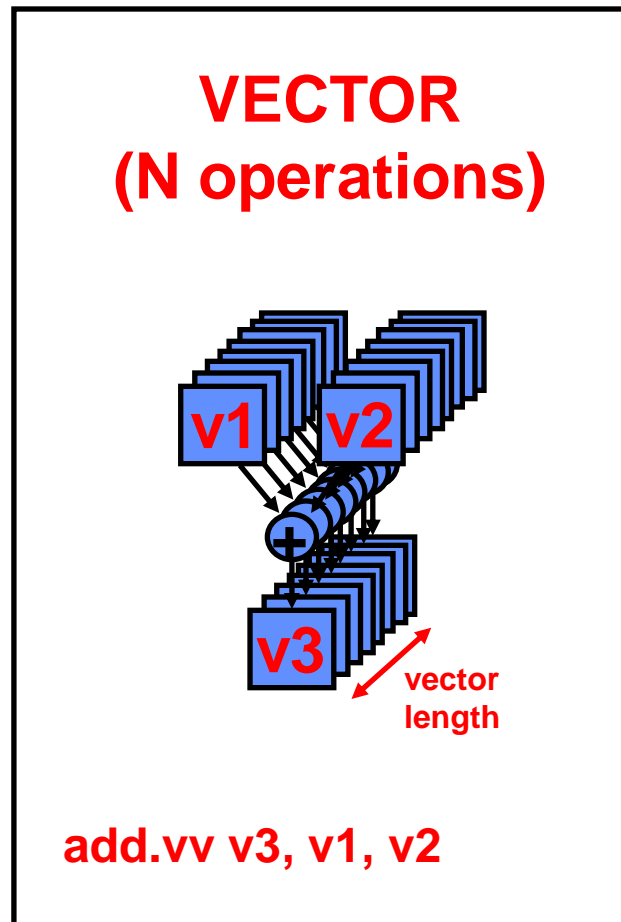
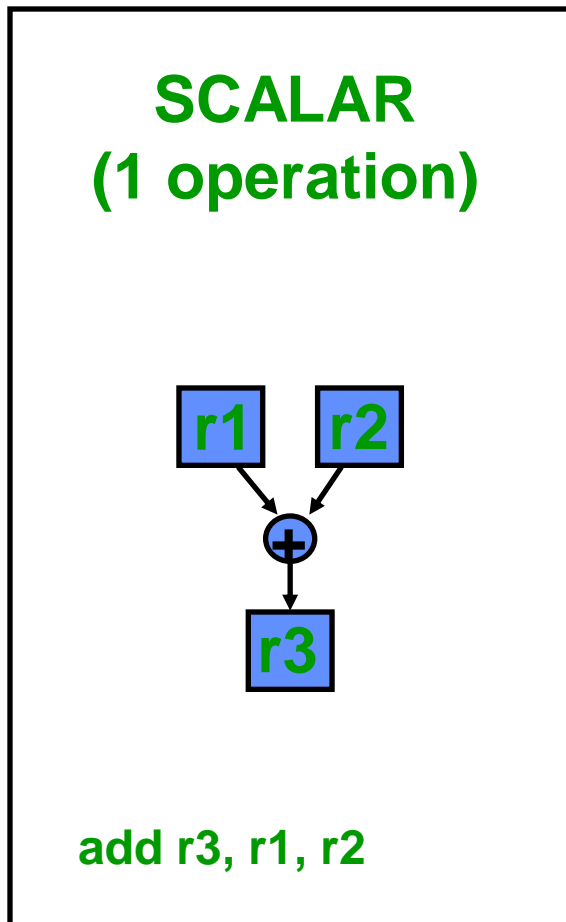
➡ Later section of this Chapter

Multiprocessors

➡ Later Chapter

Alternative Model: Vector Processing

- Vector processors have high-level operations that work on linear arrays of numbers: "vectors"



Properties of Vector Processors

- ▶ Each result independent of previous result
 - => long pipeline, compiler ensures no dependencies
 - => high clock rate
- ▶ Vector instructions access memory with known pattern
 - => highly interleaved memory
 - => amortize memory latency of over - 64 elements
 - => no (data) caches required! (Do use instruction cache)
- ▶ Reduces branches and branch problems in pipelines
- ▶ Single vector instruction implies lots of work (- loop)
 - => fewer instruction fetches

Operation & Instruction Count: RISC v. Vector Processor

(from F. Quintana, R Espasa, M Valero, An ISA comparison between superscalar and vector processors.)

Program	# basic blocks	# instructions		# vector operations	% Vect	Avg. VL
		Scalar	Vector			
Swm256	2.57	27.46	74.82	8127.98	99.7	93
Hydro2d	4.74	38.85	35.43	3684.89	99.0	101
Nasa7	16.79	139.80	55.98	3885.02	96.5	62
Su2cor	22.53	143.95	24.08	3066.07	95.5	125
Tomcatv	19.95	126.66	6.37	644.41	83.6	99
Wave5	48.99	579.77	35.88	1615.04	73.6	43

On this selection of vector-friendly SPEC92FP applications, vector reduces ops by 1.2X, instructions by 20X

Styles of Vector Architectures

- ▶ ***memory-memory vector processors***: all vector operations are memory to memory
- ▶ ***vector-register processors***: all vector operations between vector registers (except load and store)
 - ▶ Vector equivalent of load-store architectures
 - ▶ Includes all vector machines since late 1980s: Cray, Convex, Fujitsu, Hitachi, NEC
 - ▶ We assume vector-register for rest of lectures

Components of Vector Processor

- ▶ **Vector Register:** fixed length bank holding a single vector
 - ▶ has at least 2 read and 1 write ports
 - ▶ typically 8-32 vector registers, each holding 64-128 64-bit elements
- ▶ **Vector Functional Units (FUs):** fully pipelined, start new operation every clock
 - ▶ typically 4 to 8 FUs: FP add, FP mult, FP reciprocal ($1/X$), integer add, logical, shift; may have multiple of same unit
- ▶ **Vector Load-Store Units (LSUs):** fully pipelined unit to load or store a vector; may have multiple LSUs
- ▶ **Scalar registers:** single element for FP scalar or address
- ▶ **Cross-bar** to connect FUs , LSUs, registers

“DLXV” Vector Instructions

Instr.	Operands	Operation	Comment
▶ ADD<u>V</u>	V1,V2,V3	$V1 = V2 + V3$	vector + vector
▶ ADD<u>S</u>V	V1, F0 ,V2	$V1 = \mathbf{F0} + V2$	scalar + vector
▶ MULTV	V1,V2,V3	$V1 = V2 \times V3$	vector x vector
▶ MULSV	V1,F0,V2	$V1 = F0 \times V2$	scalar x vector
▶ LV	V1,R1	$V1 = M[R1..R1+63]$	load, stride=1
▶ LV<u>WS</u>	V1,R1,R2	$V1 = M[R1..R1 + \mathbf{63 * R2}]$	load, stride=R2
▶ LV<u>I</u>	V1,R1,V2	$V1 = M[R1 + \mathbf{V2i}, i=0..63]$	indir.("gather")
▶ CeqV	VM,V1,V2	$VMASK_i = (V1_i = V2_i)?$	comp. setmask
▶ MOV	<u>VLR</u> ,R1	Vec. Len. Reg. = R1	set vector length
▶ MOV	<u>VM</u> ,R1	Vec. Mask = R1	set vector mask

Memory operations

▶ Load/store operations move groups of data between registers and memory

▶ Three types of addressing

▶ Unit stride

- Fastest

▶ Non-unit (constant) stride

▶ Indexed (gather-scatter)

- Vector equivalent of register indirect
- Good for sparse arrays of data
- Increases number of programs that vectorize

DAXPY ($Y = \underline{a} * \underline{X} + Y$)

Assuming vectors X, Y
are length 64

Scalar vs. **Vector**

```
LD    F0,a      ;load scalar a
LV    V1,Rx     ;load vector X
MULTS V2,F0,V1  ;vector-scalar mult.
LV    V3,Ry     ;load vector Y
ADDV  V4,V2,V3  ;add
SV    Ry,V4     ;store the result
```

```
LD    F0,a
ADDI  R4,Rx,#512 ;last address to load
loop: LD    F2, 0(Rx) ;load X(i)
      MULTD F2,F0,F2 ;a*X(i)
      LD    F4, 0(Ry) ;load Y(i)
      ADDD  F4,F2,F4 ;a*X(i) + Y(i)
      SD    F4,0(Ry) ;store into Y(i)
      ADDI  Rx,Rx,#8 ;increment index to X
      ADDI  Ry,Ry,#8 ;increment index to Y
      SUB   R20,R4,Rx ;compute bound
      BNZ  R20,loop ;check if done
```

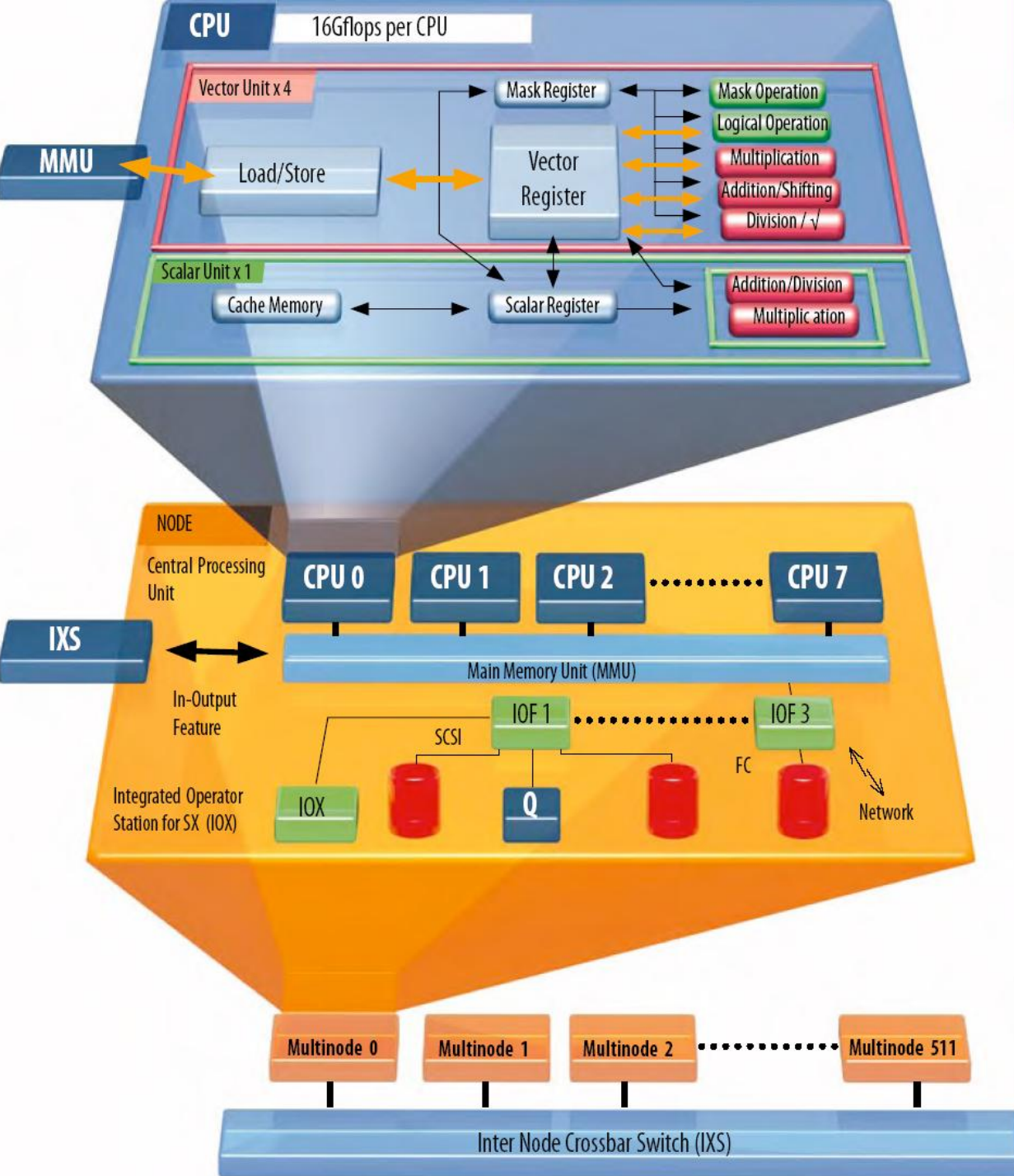
**578 (2+9*64) vs.
321 (1+5*64) ops (1.8X)**

**578 (2+9*64) vs.
6 instructions (96X)**

**64 operation vectors +
no loop overhead**

**also 64X fewer pipeline
hazards**

NEC SX-8



▶ 35 GFLOPs peak per CPU

▶ Eg University of Stuttgart installation:

- ▶ Peak Performance 12 TFlops
- ▶ 72 nodes, 8 CPUs per node
- ▶ Memory 9.2 TB
- ▶ Disk 160 TB shared disk, 72 * 140 GB local
- ▶ 16GB/s node-to-node interconnect

<http://www.hlr.de/systems/platforms/nec-sx8/>

Vector Length

- What to do when vector length is not exactly 64?
- vector-length register*** (VLR) controls the length of any vector operation, including a vector load or store.
(cannot be > the length of vector registers)

do 10 i = 1, n

10 $Y(i) = a * X(i) + Y(i)$

- Don't know n until runtime!
n > Max. Vector Length (MVL)?

Strip Mining

- Suppose Vector Length $>$ Max. Vector Length (MVL)?
- **Strip mining**: generation of code such that each vector operation is done for a size \check{S} to the MVL
- 1st loop do short piece ($n \bmod \text{MVL}$), rest $\text{VL} = \text{MVL}$

```
low = 1
VL = (n mod MVL) /*find the odd size piece*/
do 1 j = 0,(n / MVL) /*outer loop*/
    do 10 i = low,low+VL-1 /*runs for length VL*/
        Y(i) = a*X(i) + Y(i) /*main operation*/
10 continue
low = low+VL /*start of next vector*/
VL = MVL /*reset the length to max*/
1 continue
```

Common Vector Metrics

 **R_{∞}** : MFLOPS rate on an infinite-length vector

- ➔ vector “speed of light”
- ➔ Real problems do not have unlimited vector lengths, and the start-up penalties encountered in real problems will be larger
- ➔ (R_n is the MFLOPS rate for a vector of length n)

 **$N_{1/2}$** : The vector length needed to reach one-half of R_{∞} .

- ➔ a good measure of the impact of start-up

 **N_V** : The vector length needed to make vector mode faster than scalar mode

- ➔ measures both start-up and speed of scalars relative to vectors, quality of connection of scalar unit to vector unit



Vector Stride

- Suppose adjacent elements not sequential in memory

do 10 i = 1,100

do 10 j = 1,100

A(i,j) = 0.0

do 10 k = 1,100

10 A(i,j) = A(i,j)+B(i,k)*C(k,j)

- Either B or C accesses not adjacent (800 bytes between)
- stride**: distance separating elements that are to be merged into a single vector (caches do unit stride)
=> **LVWS** (load vector with stride) instruction
- Strides => can cause bank conflicts
(e.g., stride = 32 and 16 banks)
- Think of address per vector element

Vector Opt #1: Chaining

Suppose:

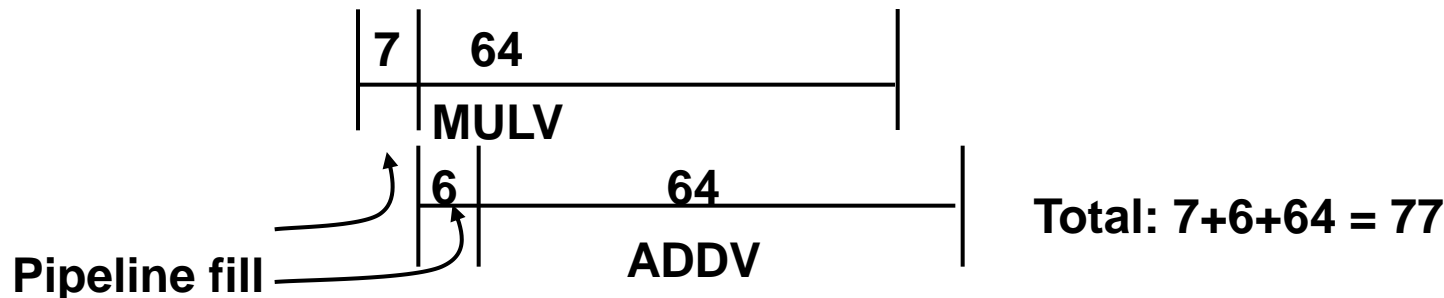
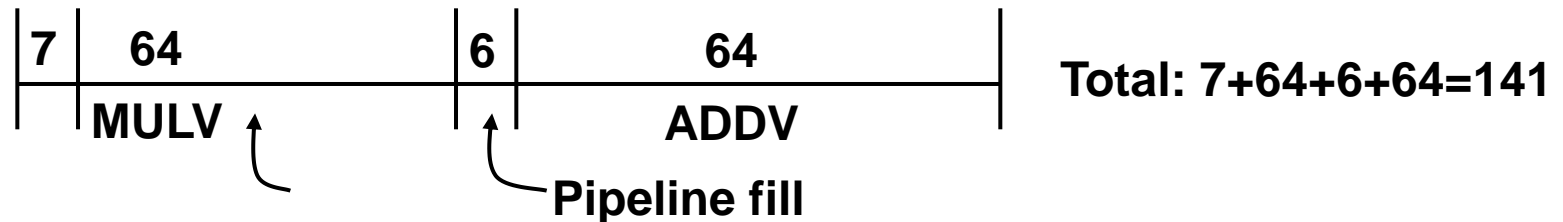
MULV V1, V2, V3

ADDV V4, V1, V5 ; separate convoy?

chaining: vector register (V1) is not as a single entity but as a group of individual registers, then pipeline forwarding can work on individual elements of a vector

Flexible chaining: allow vector to chain to any other active vector operation => more read/write port

As long as enough HW, increases convoy size



Vector Opt #2: Conditional Execution

▶ Suppose:

```
do 100 i = 1, 64
```

```
    if (A(i) .ne. 0) then
```

```
        A(i) = A(i) – B(i)
```

```
    endif
```

```
100 continue
```

▶ ***vector-mask control*** takes a Boolean vector: when ***vector-mask register*** is loaded from vector test, vector instructions operate only on vector elements whose corresponding entries in the vector-mask register are 1.

▶ Still requires clock even if result not stored; if still performs operation, what about divide by 0?



Vector mask utilisation

(from F. Quintana, R Espasa, M Valero, An ISA comparison between superscalar and vector processors.)

Program	Instructions executed under vector mask		Operations executed under vector mask	
	Total Number (x10 ⁶)	% over total instructions	Total Number (x10 ⁶)	% over total operations
Swm256	0.01	0.015	0.13	0.016
Hydro2d	5.75	7.750	582.91	15.650
Nasa7	0.07	0.036	8.02	0.20
Su2cor	1.06	0.630	130.75	3.95
Tomcatv	0.00	0.000	0.00	0.00
Wave5	5.17	0.840	80.00	3.64

- ▶ Many apps don't need, essential for a few
- ▶ Actually it's a form of speculation...

Vector Opt #3: Sparse Matrices

➤ Suppose:

do 100 i = 1,n

100 A(K(i)) = A(K(i)) + C(M(i))

➤ **gather** (LVI) operation takes an **index vector** and fetches the vector whose elements are at the addresses given by adding a base address to the offsets given in the index vector => **a nonsparse vector in a vector register**

➤ After these elements are operated on in dense form, the sparse vector can be stored in expanded form by a **scatter** store (SVI), using the same index vector

➤ Can't be done by compiler since can't know K_i elements distinct, no dependencies; by compiler directive

➤ Use CVI to create index 0, 1xm, 2xm, ..., 63xm



Vector for Multimedia?

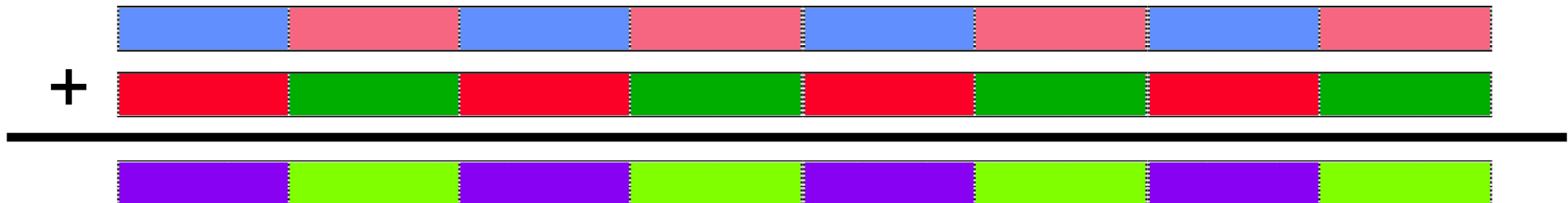
Intel MMX/SSE instruction set extensions

- ➔ Similar extensions on other processor families, eg PowerPC AltiVec

Idea: pack multiple short-word operands into one long register

- ➔ Eg 128-bit register

- 2 64-bit doubles
- 4 32-bit floats or ints
- 8 16-bit ints or fixed-point
- 16 8-bit ints
- Often with media-specific instructions eg saturated arithmetic



Claim: overall speedup 1.5 to 2X for 2D/3D graphics, audio, video, speech, comm., ...

- ➔ Initially hand-coded, accessible using special intrinsic functions
- ➔ Delivered via libraries such as the Intel Performance Primitives (IPP)
- ➔ Some support from compilers such as Intel's, but awkward constraints (eg alignment of operands)



Vector Pitfalls

- ▶ **Pitfall: Concentrating on peak performance and ignoring start-up overhead: N_v (length faster than scalar) > 100!**
- ▶ **Pitfall: Increasing vector performance, without comparable increases in scalar performance (Amdahl's Law)**
 - ▶ failure of Cray competitor from his former company
- ▶ **Pitfall: Good processor vector performance without providing good memory bandwidth**
 - ▶ MMX?

Vector Advantages

- ▶ Easy to get high performance; N operations:
 - ▶ are independent
 - ▶ use same functional unit
 - ▶ access disjoint registers
 - ▶ access registers in same order as previous instructions
 - ▶ access contiguous memory words or known pattern
 - ▶ can exploit large memory bandwidth
 - ▶ hide memory latency (and any other latency)
- ▶ Scalable (get higher performance as more HW resources available)
- ▶ Compact: Describe N operations with 1 short instruction (v. VLIW)
- ▶ Predictable (real-time) performance vs. statistical performance (cache)
- ▶ Multimedia ready: choose $N * 64b$, $2N * 32b$, $4N * 16b$, $8N * 8b$
- ▶ Mature, developed compiler technology



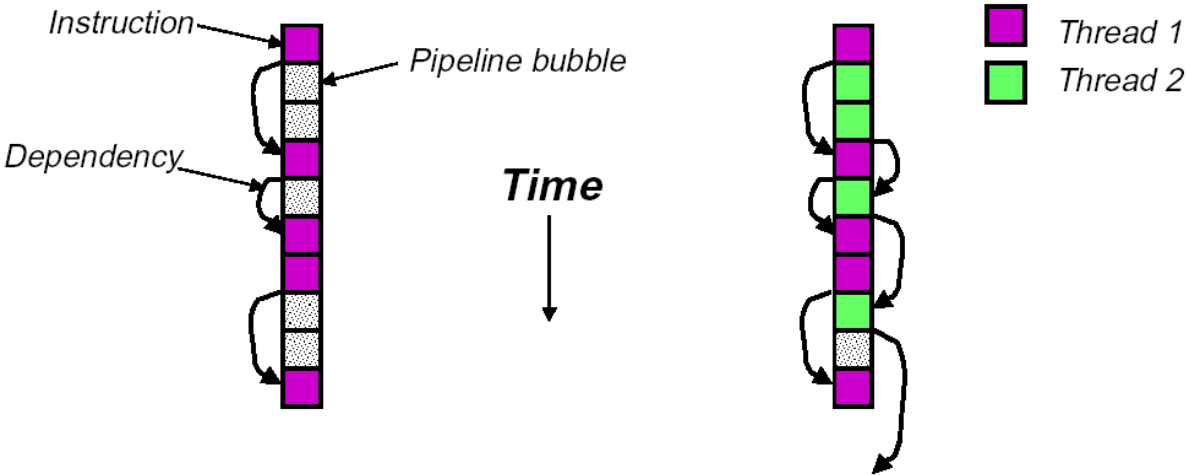
Vector Summary

- ▶ **Alternate model accommodates long memory latency, doesn't rely on caches as does Out-Of-Order, superscalar/VLIW designs**
- ▶ **Much easier for hardware: more powerful instructions, more predictable memory accesses, fewer hazards, fewer branches, fewer mispredicted branches, ...**
- ▶ **What % of computation is vectorizable?**
- ▶ **Is vector a good match to new apps such as multimedia, DSP?**

Beyond ILP: Multithreading, Simultaneous Multithreading (SMT)

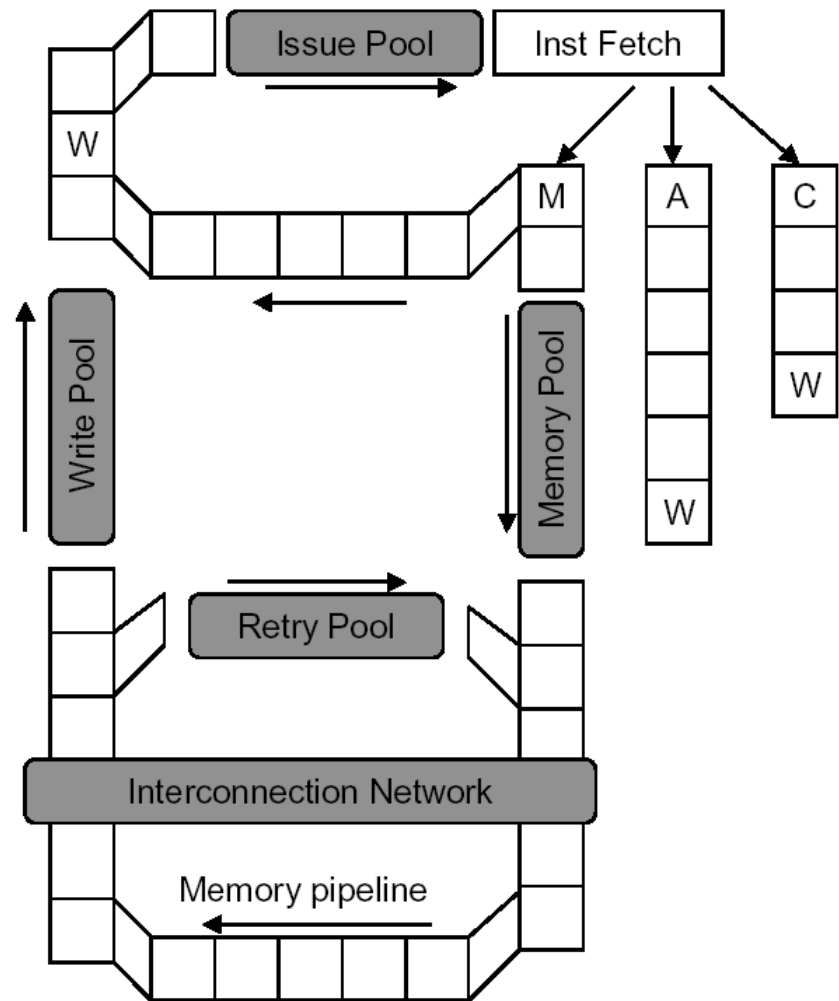
▶ Cray/Tera MTA

- ▶ <http://www.cray.com/products/systems/mta/>,
- ▶ <http://www.utc.edu/~jdumas/cs460/papers/fa01/craymta/>

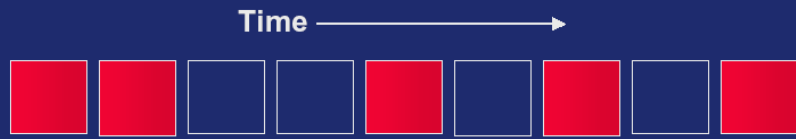


- **Problem:** Dependencies limit sustainable throughput of single instruction stream
- **Solution:** Interleave execution of two or more instruction streams on same hardware to increase utilization

(Source: Asanovic <http://www.cag.lcs.mit.edu/6.893-f2000/lectures/l06-tera.pdf>)

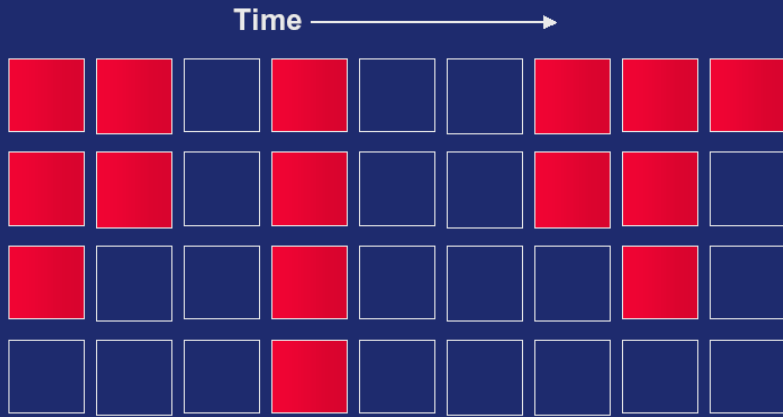


Instruction Issue



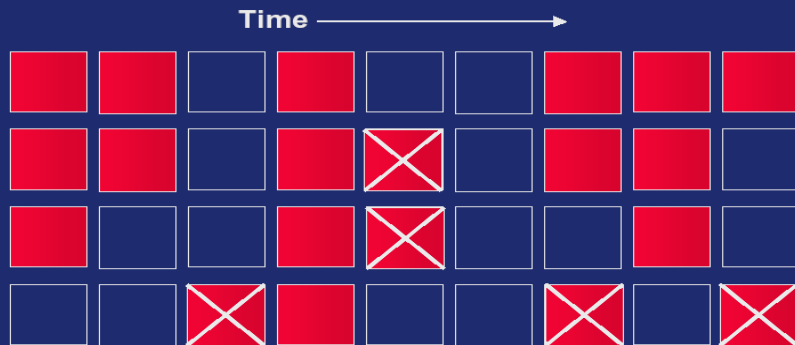
Reduced function unit utilization due to dependencies

Superscalar Issue



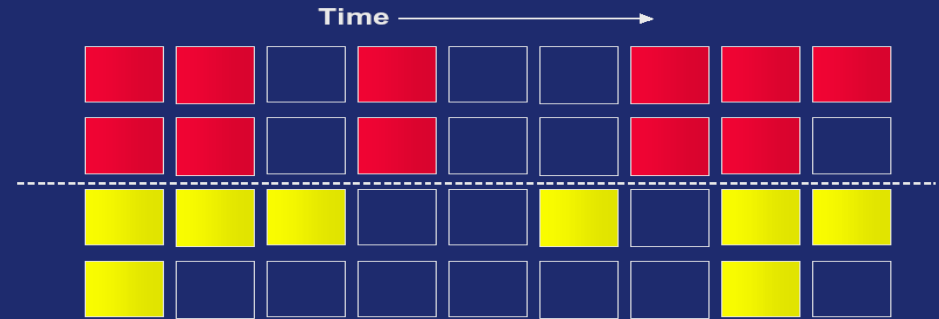
Superscalar leads to more performance, but lower utilization

Predicated Issue



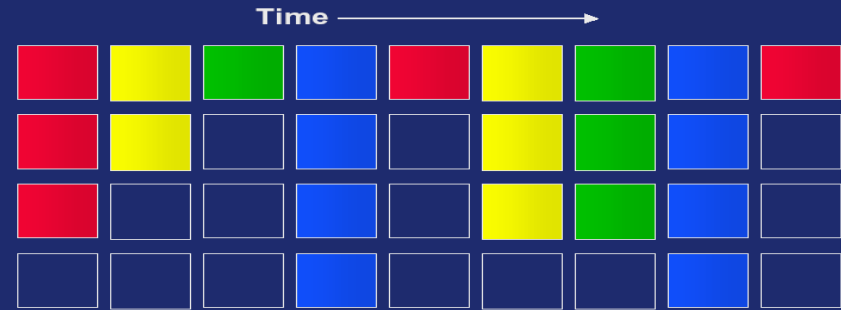
Adds to function unit utilization, but results are thrown away

Chip Multiprocessor



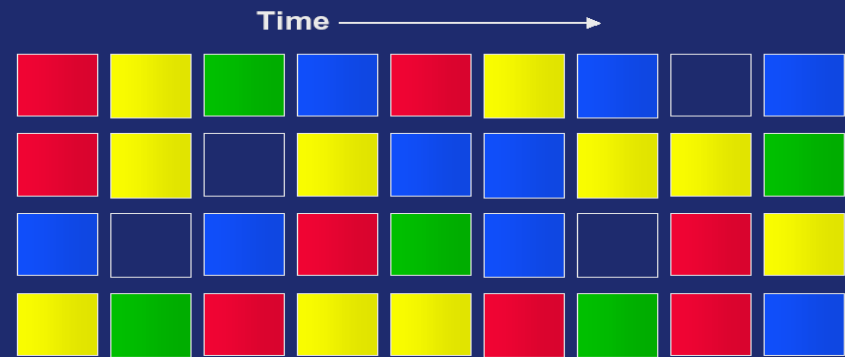
Limited utilization when only running one thread

Fine Grained Multithreading



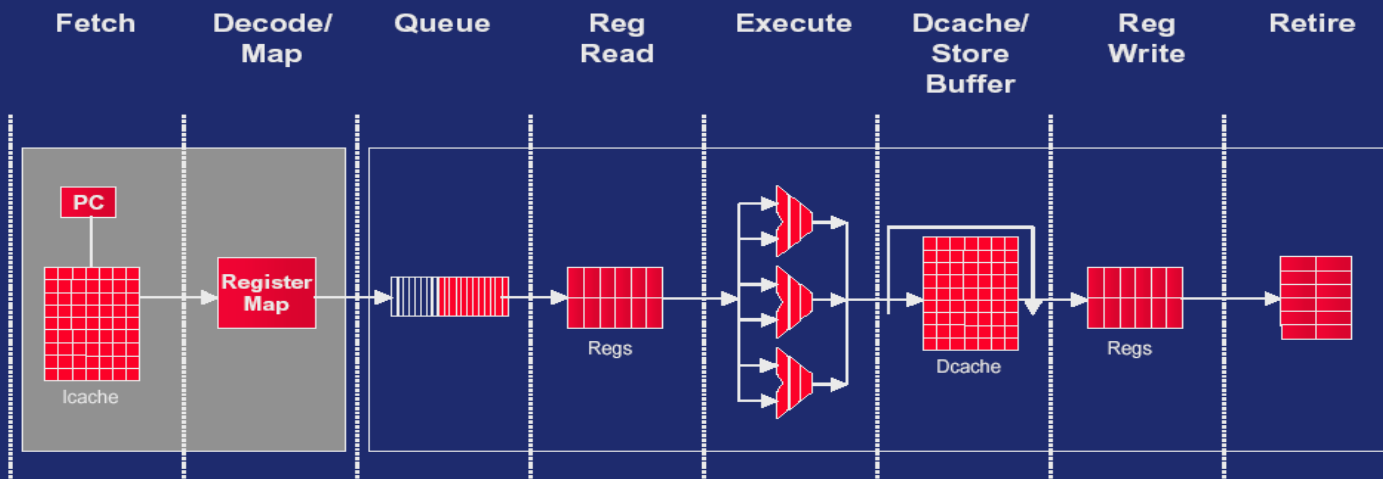
Intra-thread dependencies still limit performance

Simultaneous Multithreading



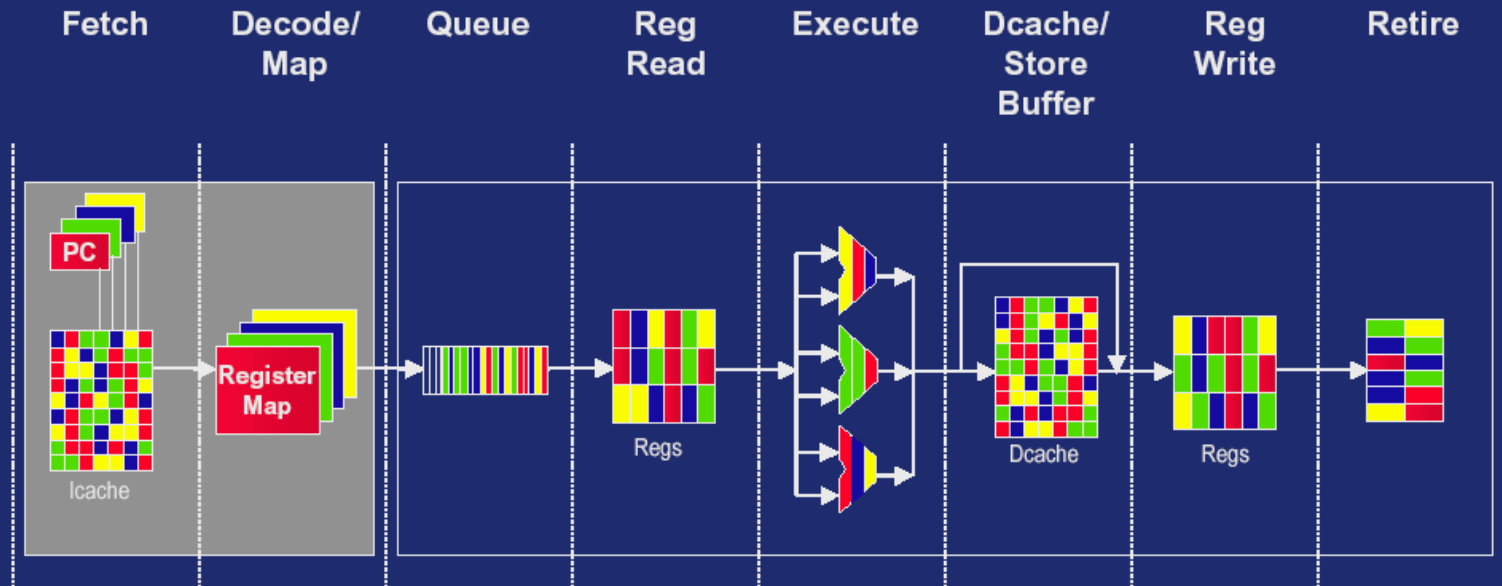
Maximum utilization of function units by independent operations

Basic Out-of-order Pipeline



SMT

SMT Pipeline



Alpha 21464

One CPU with 4 Thread Processing Units (TPUs)

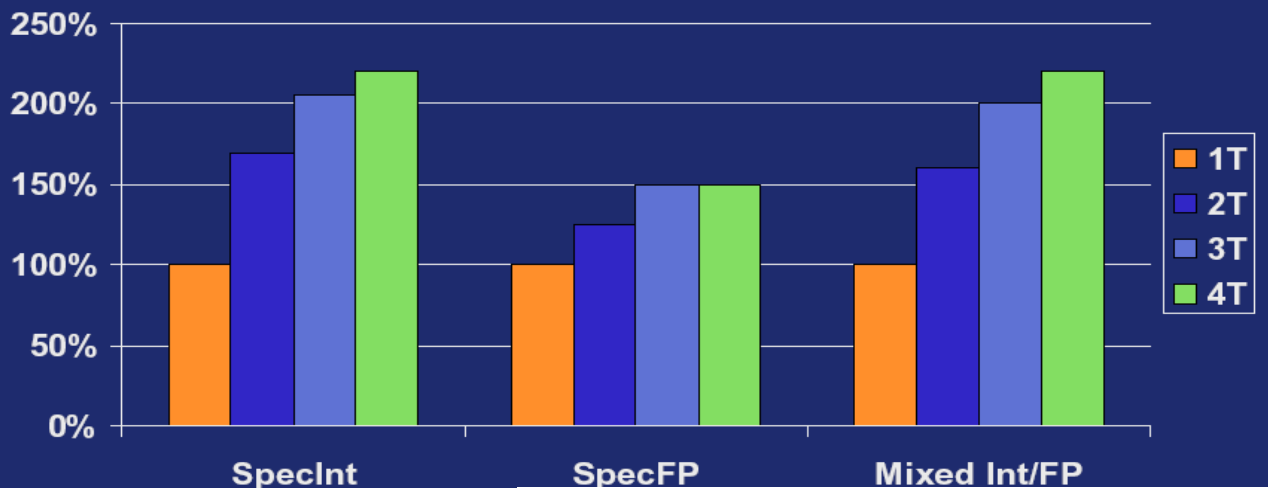
“6% area overhead over single-thread 4-issue CPU”

Multiprogrammed workload

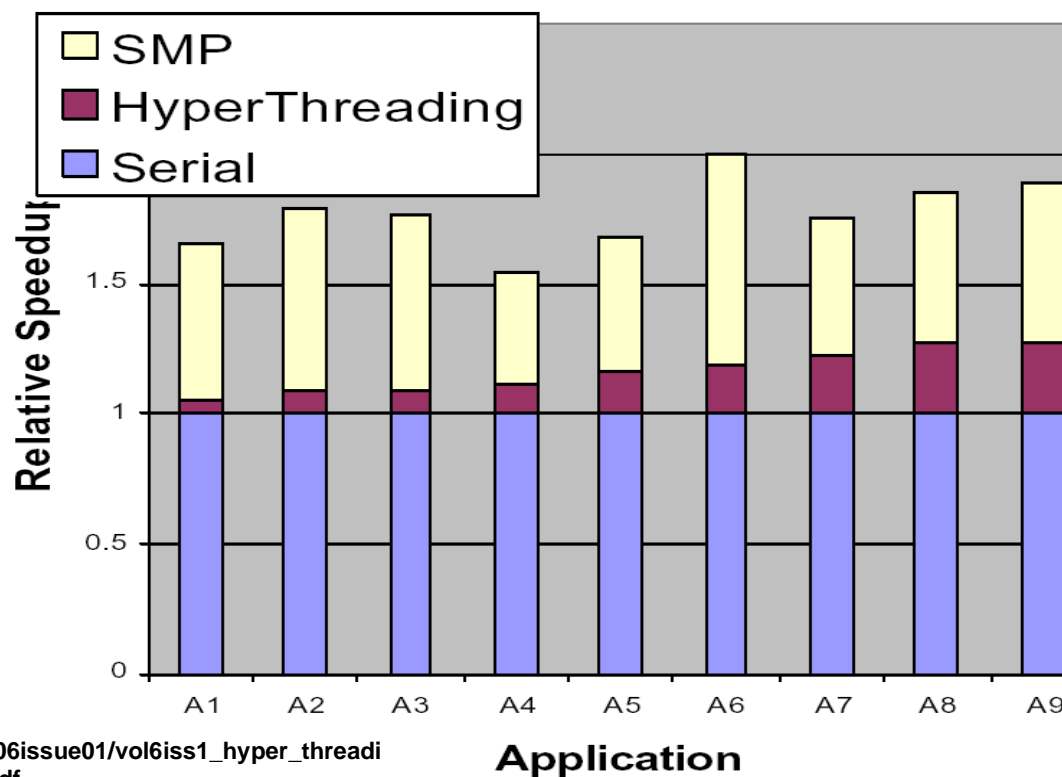
SMT

performance

Alpha 21464



Intel Pentium 4 with hyperthreading:

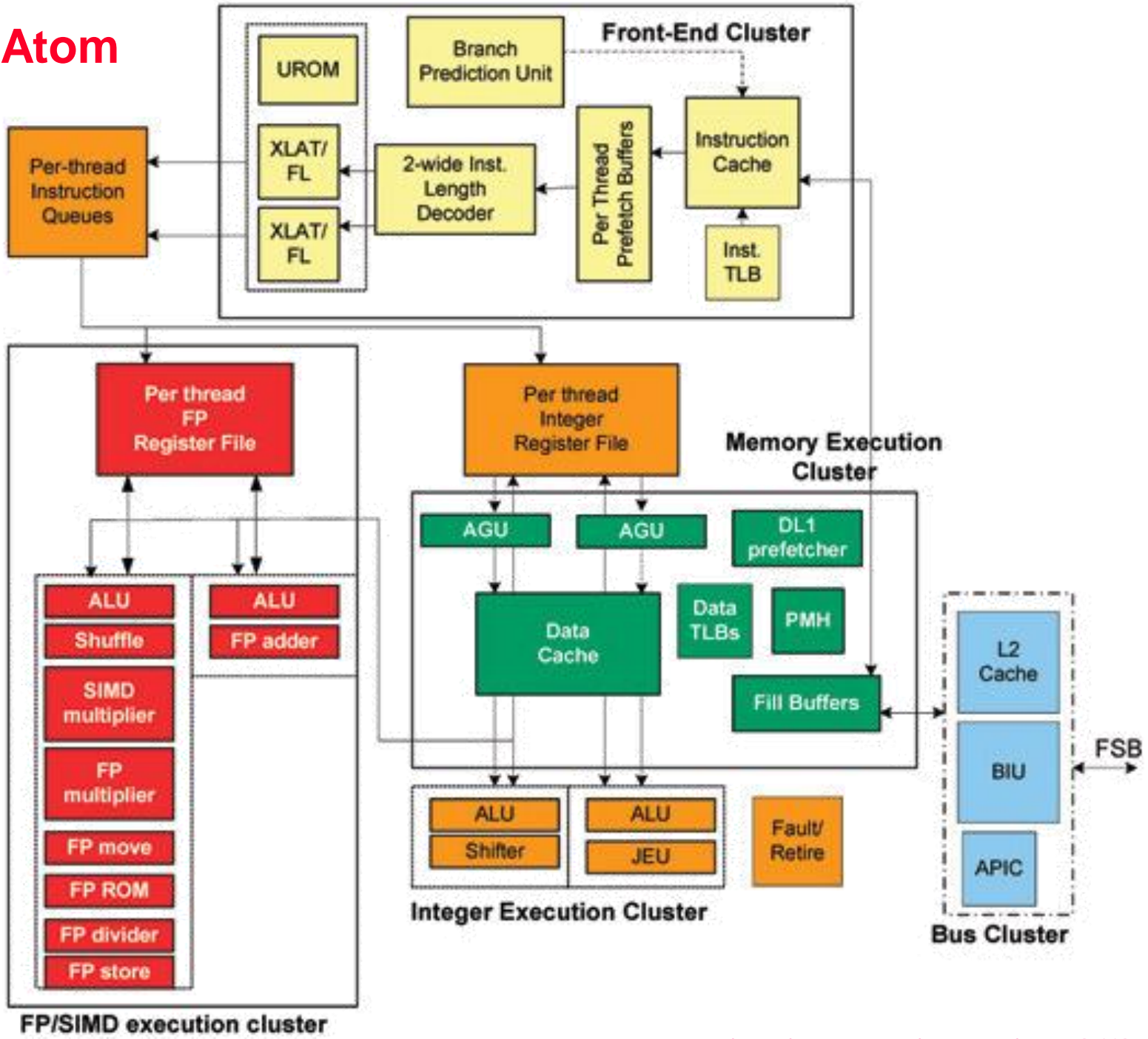


Code	Description
A1	Mechanical Design Analysis (finite element method) This application is used for metal-forming, drop testing, and crash simulation.
A2	Genetics A genetics application that correlates DNA samples from multiple animals to better understand congenital diseases.
A3	Computational Chemistry This application uses the self-consistent field method to compute chemical properties of molecules such as new pharmaceuticals.
A4	Mechanical Design Analysis This application simulates the metal-stamping process.
A5	Mesoscale Weather Modeling This application simulates and predicts mesoscale and regional-scale atmospheric circulation.
A6	Genetics This application is designed to generate Expressed Sequence Tags (EST) clusters, which are used to locate important genes.
A7	Computational Fluid Dynamics This application is used to model free-surface and confined flows.
A8	Finite Element Analysis This finite element application is specifically targeted toward geophysical engineering applications.
A9	Finite Element Analysis This explicit time-stepping application is used for crash test studies and computational fluid dynamics.

SMT in the Intel Atom

- Intel's bid to steal back some of the low-power market for IA-32 and Windows

- In-order
- 2-way SMT



Conclusion

▶ 1985-2000: 1000X performance

▶ Moore's Law transistors/chip => Moore's Law for Performance/MPU

▶ “industry been following a roadmap of ideas known in 1985 to exploit Instruction Level Parallelism and (real) Moore's Law to get 1.55X/year” (John Hennessy)

▶ Caches, Pipelining, Superscalar, Branch Prediction, Out-of-order execution,
...

▶ ILP limits: To make performance progress in future need to have explicit parallelism from programmer vs. implicit parallelism of ILP exploited by compiler, HW?

▶ Impact on you: if you care about performance, better think about explicitly parallel algorithms vs. rely on ILP?