
CS 5513 Computer Architecture

Lecture 5 – Instruction Level Parallelism

Outline

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

Recall from Pipelining Review

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

Instruction Level Parallelism

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

Instruction-Level Parallelism (ILP)

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

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

Loop-Level Parallelism

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

Data Dependence and Hazards


- Instr_j is **data dependent** (aka **true dependence**) on Instr_i :
 1. Instr_j tries to read operand before Instr_i writes it
 - $\begin{array}{l} \text{I: add } \textcolor{red}{r1}, r2, r3 \\ \text{J: sub } r4, \textcolor{red}{r1}, r3 \end{array}$
 2. or Instr_j is data dependent on Instr_k which is dependent on Instr_i
- If two instructions are data dependent, they cannot execute simultaneously or be completely overlapped
- Data dependence in instruction sequence \Rightarrow data dependence in source code \Rightarrow effect of original data dependence must be preserved
- If data dependence caused a hazard in pipeline, called a **Read After Write (RAW) hazard**

ILP and Data Dependencies, Hazards

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

Name Dependence #1: Anti-dependence

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


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

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

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

Name Dependence #2: Output dependence

- Instr_j writes operand before Instr_i writes it.

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

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

Control Dependencies

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

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

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

Control Dependence Ignored

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

Exception Behavior

- Preserving exception behavior
⇒ any changes in instruction execution order must not change how exceptions are raised in program
(⇒ no new exceptions)
- Example:
 DADDU R2 , R3 , R4
 BEQZ R2 , L1
 LW R1 , 0 (R2)
L1 :
 – (Assume branches not delayed)
- Problem with moving LW before BEQZ?

Data Flow

- **Data flow**: actual flow of data values among instructions that produce results and those that consume them
 - branches make flow dynamic, determine which instruction is supplier of data

- **Example:**

DADDU R1, R2, R3

BEQZ R4, L

DSUBU R1, R5, R6

L: ...

OR R7, R1, R8

- OR depends on DADDU or DSUBU?
Must preserve data flow on execution

Outline

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

Software Techniques - Example

- This code, add a scalar to a vector:

```
for (i=1000; i>0; i=i-1)  
    x[i] = x[i] + s;
```
- Assume following latencies for all examples
 - Ignore delayed branch in these examples

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

FP Loop: Where are the Hazards?

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

```
Loop: L.DF0,0(R1) ;F0=vector element
      ADD.D F4,F0,F2 ;add scalar from F2
      S.D0(R1),F4 ;store result
      DADDUIR1,R1,-8 ;decrement pointer 8B (DW)
      BNEZ R1,Loop ;branch R1!=zero
```

FP Loop Showing Stalls

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

*Instruction Latency in
producing result using result clock cycles*

FP ALU op Another FP ALU op 3

FP ALU op Store double 2

Load doubleFP ALU op 1

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

Revised FP Loop Minimizing Stalls

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

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

*Instruction Instruction Latency in
producing result using result clock cycles*

FP ALU op Another FP ALU op 3


FP ALU op Store double 2

Load double FP ALU op 1

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

Unroll Loop Four Times (straightforward way)

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



**Rewrite loop to
minimize stalls?**

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

Unrolled Loop Detail

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

Unrolled Loop That Minimizes Stalls

```
1 Loop:    L.D F0,0(R1)
2    L.D F6,-8(R1)
3    L.D F10,-16(R1)
4    L.D F14,-24(R1)
5    ADD.D  F4,F0,F2
6    ADD.D  F8,F6,F2
7    ADD.D  F12,F10,F2
8    ADD.D  F16,F14,F2
9    S.D 0(R1),F4
10   S.D -8(R1),F8
11   S.D -16(R1),F12
12   DSUBUI R1,R1,#32
13   S.D 8(R1),F16 ; 8-32 = -24
14   BNEZ   R1,LOOP
```

14 clock cycles, or 3.5 per iteration

5 Loop Unrolling Decisions

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

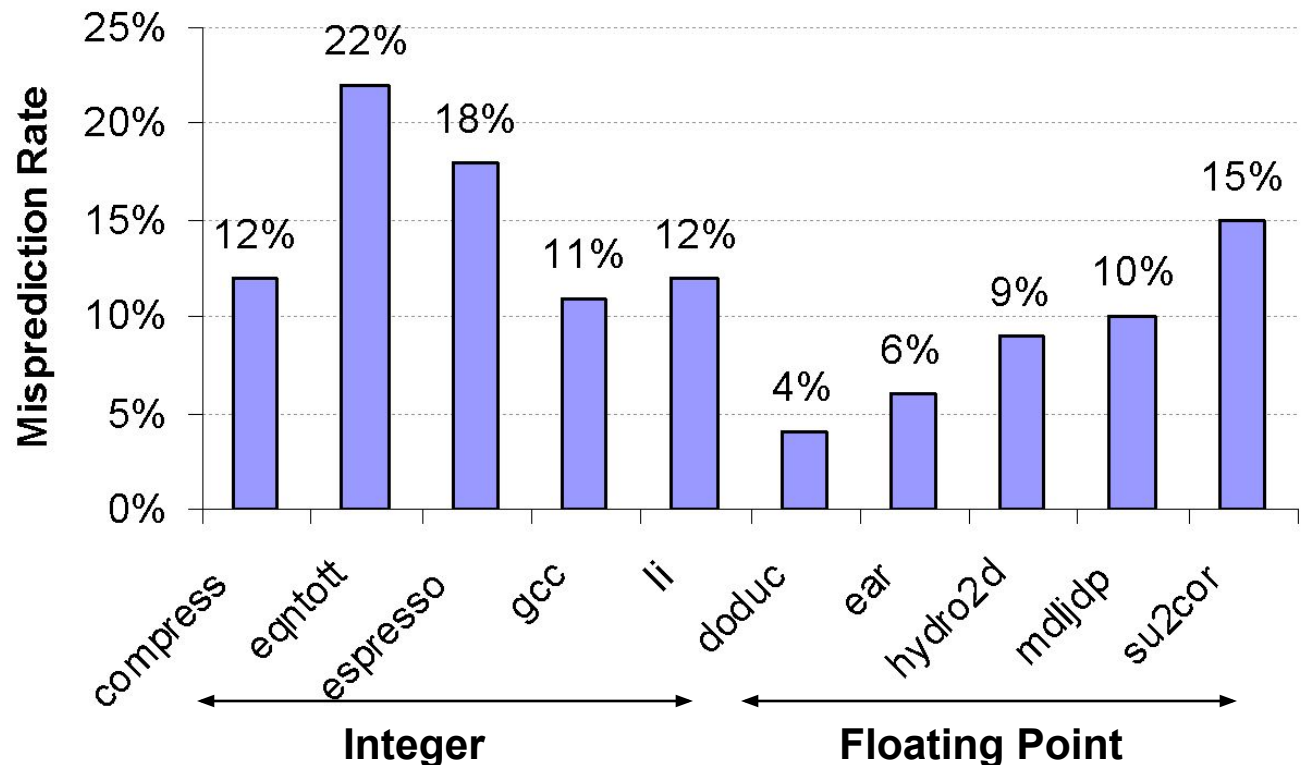
3 Limits to Loop Unrolling

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

Static Branch Prediction

- Previous lecture showed scheduling code around delayed branch
- To reorder code around branches, need to predict branch statically when compile
- Simplest scheme is to predict a branch as taken
 - Average misprediction = untaken branch frequency = 34% SPEC

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



Dynamic Branch Prediction

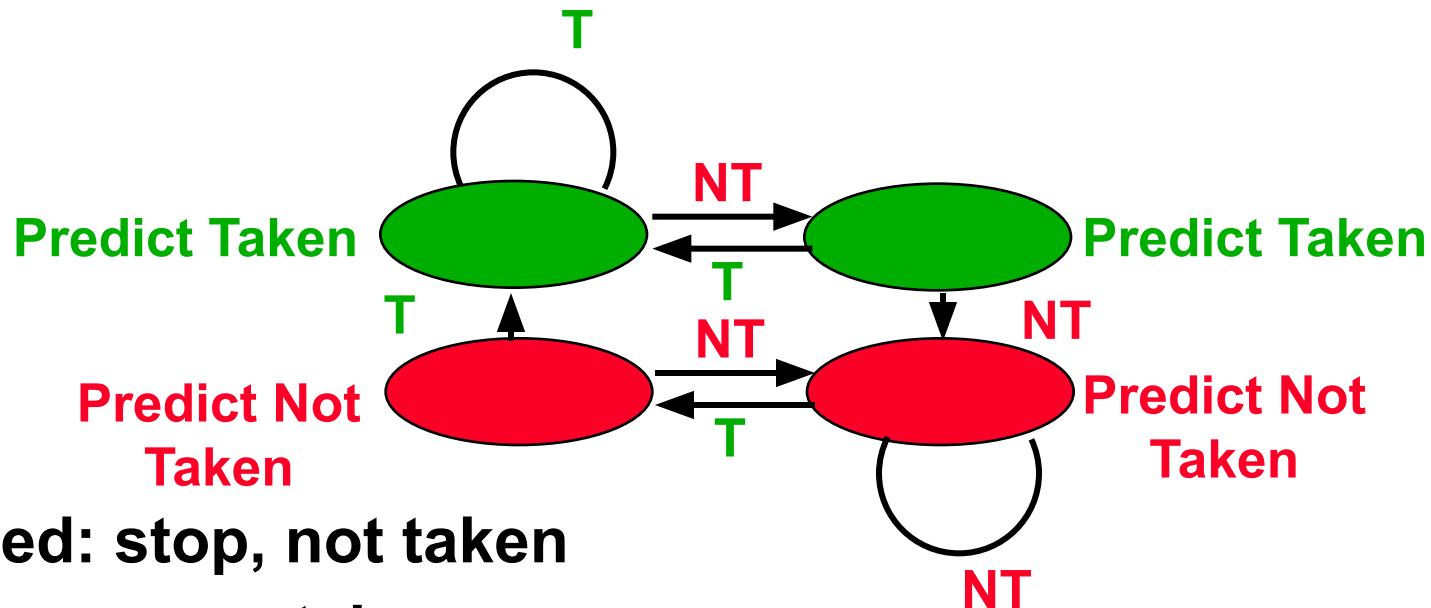
- **Why does prediction work?**
 - Underlying algorithm has regularities
 - Data that is being operated on has regularities
 - Instruction sequence has redundancies that are artifacts of way that humans/compiler think about problems
- **Is dynamic branch prediction better than static branch prediction?**
 - Almost always yes
 - There are a small number of important branches in programs which have dynamic behavior

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
- Problem: in a loop, 1-bit BHT will cause two mispredictions (avg is 9 iterations before exit):
 - End of loop case, when it exits instead of looping as before
 - First time through loop on *next* time through code, when it predicts exit instead of looping

Dynamic Branch Prediction

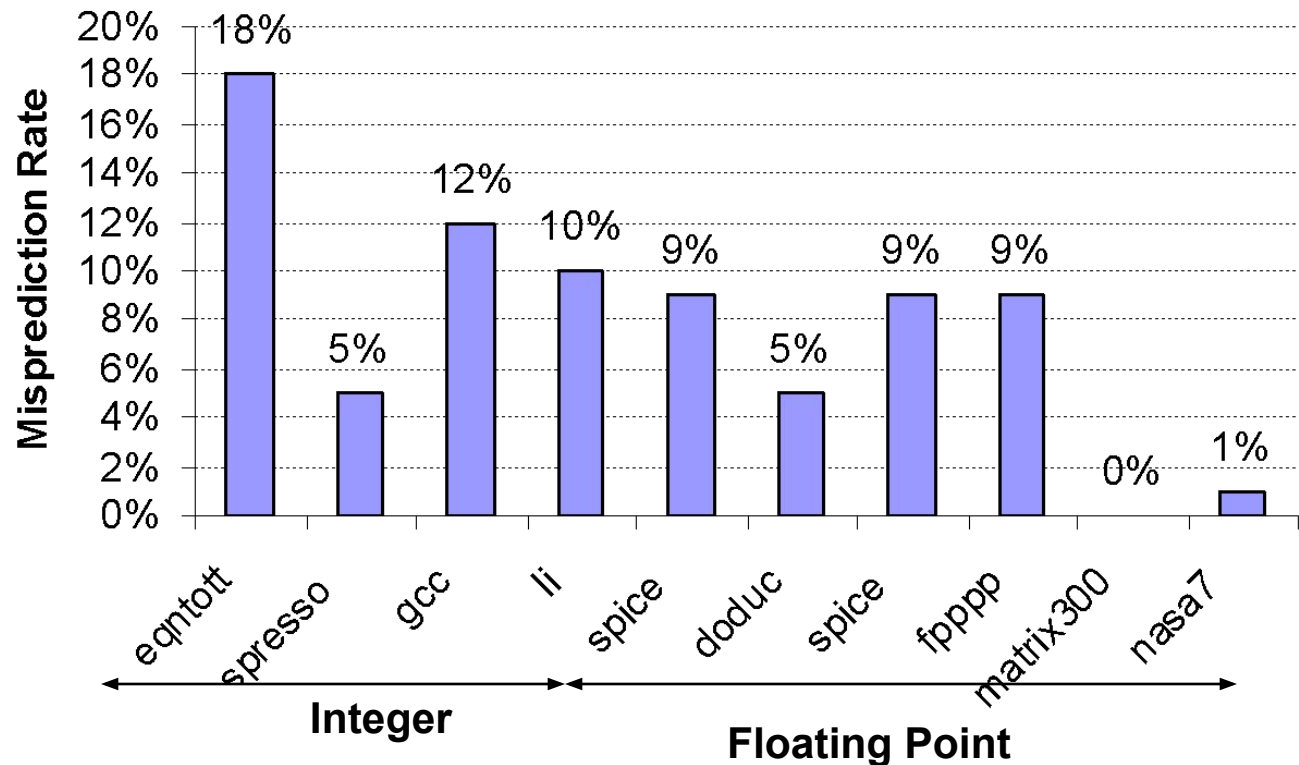
- Solution: 2-bit scheme where change prediction only if get misprediction *twice*



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

BHT Accuracy

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



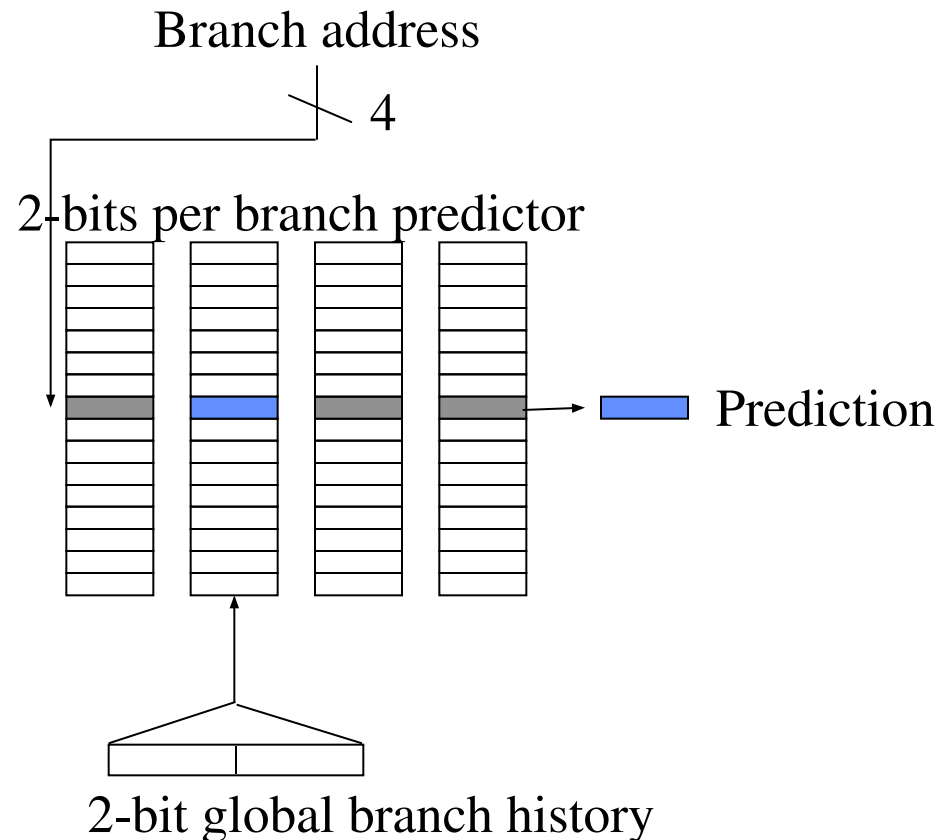
Correlated Branch Prediction

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

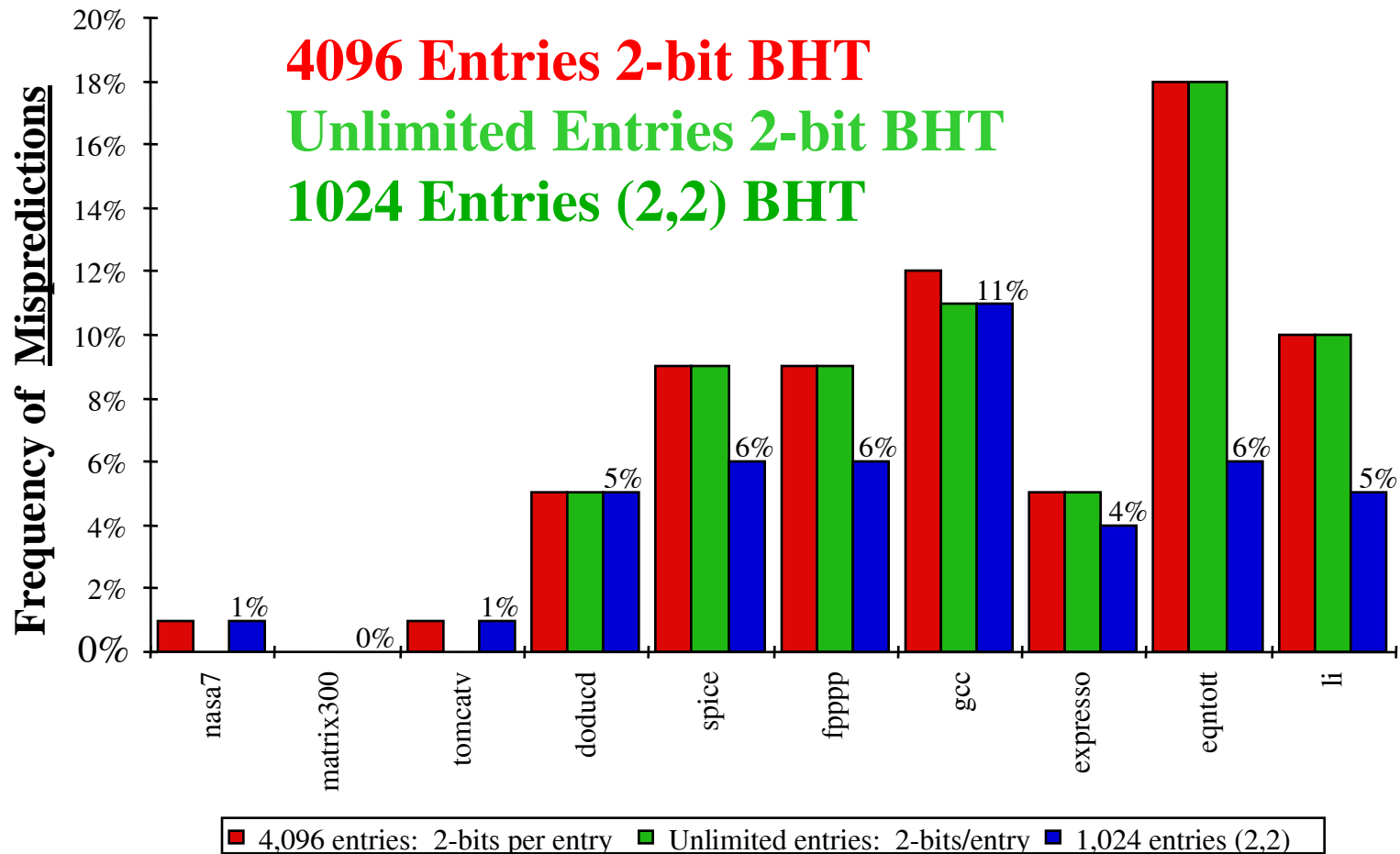
Correlating Branches

(2,2) predictor

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

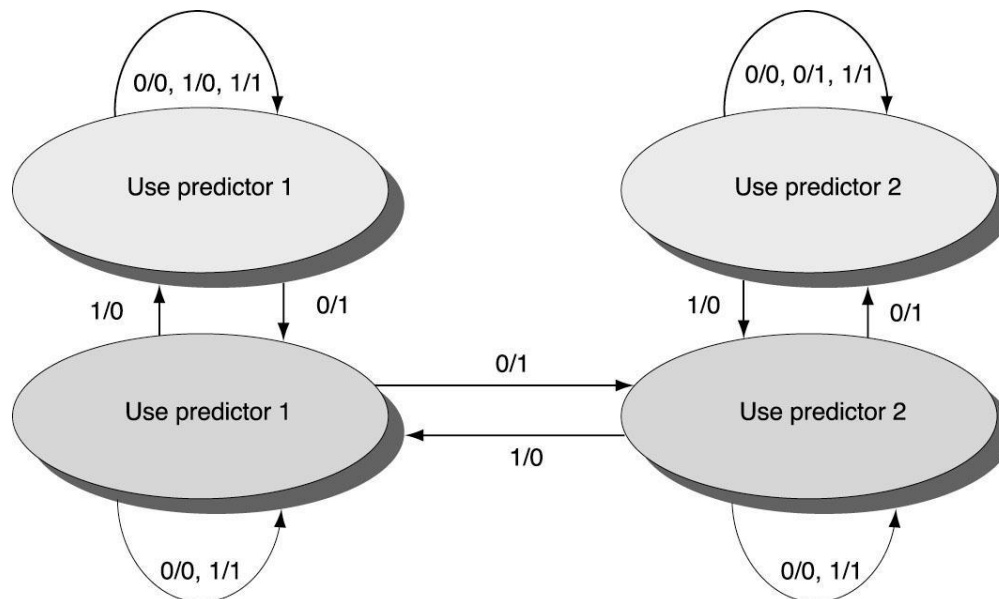


Accuracy of Different Schemes



Tournament Predictors

- Multilevel branch predictor
- Use n -bit saturating counter to choose between [predictors](#)
- Usual choice between global and local predictors



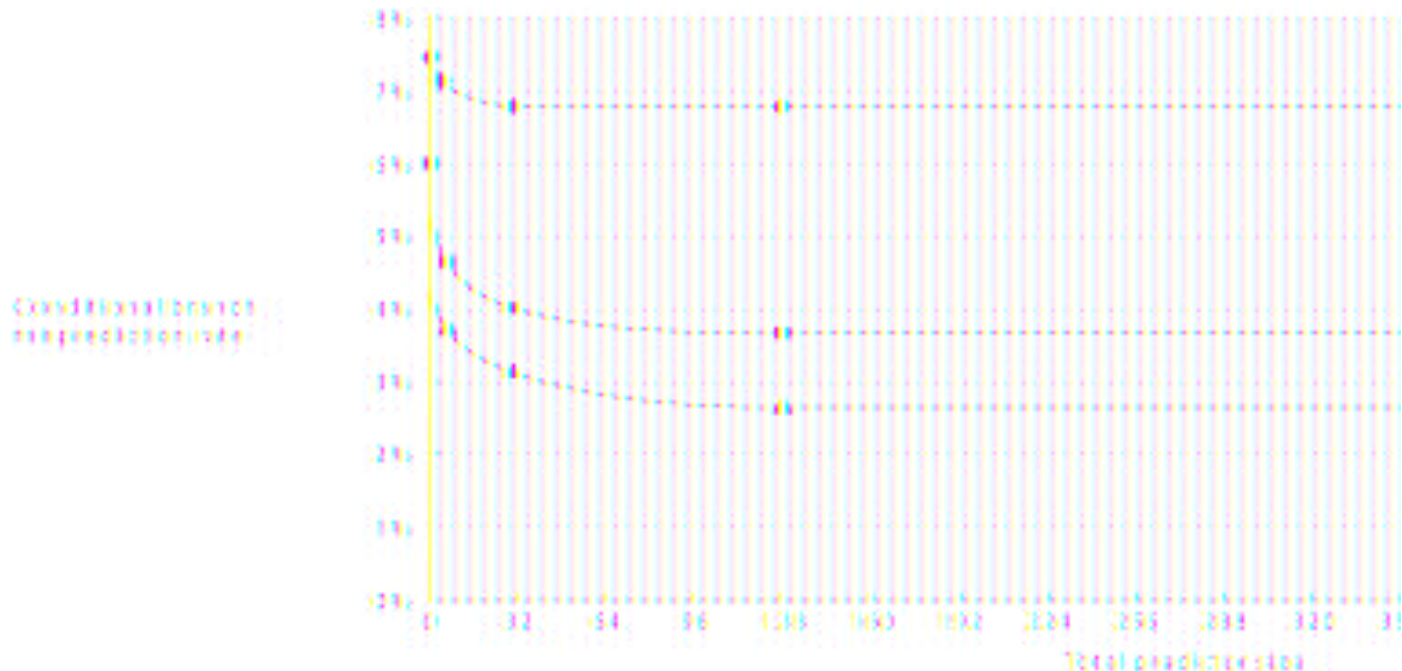
Tournament Predictors

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

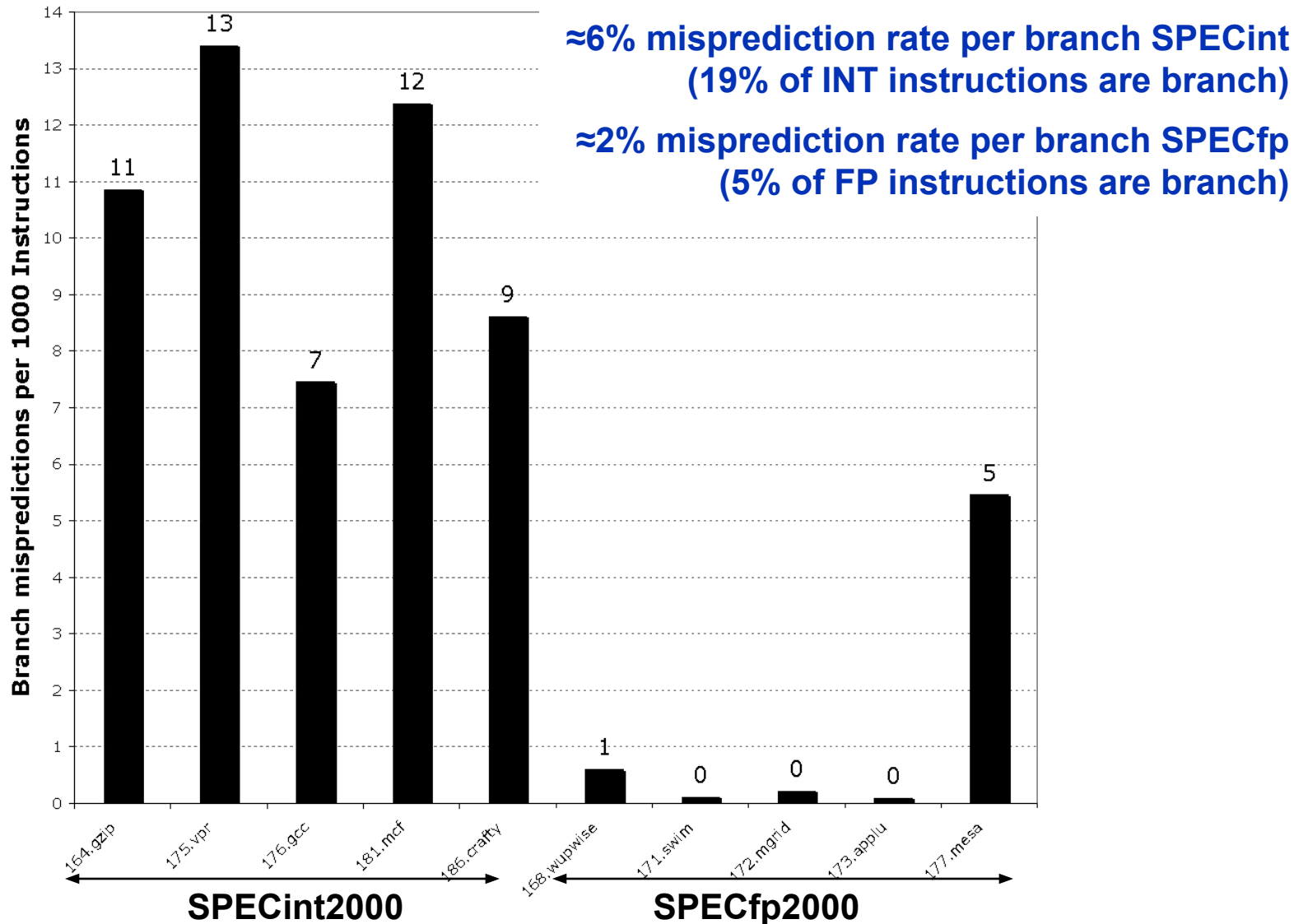
- **Global predictor**
 - 4K entries index by history of last 12 branches ($2^{12} = 4K$)
 - Each entry is a standard 2-bit predictor
- **Local predictor**
 - Local history table: 1024 10-bit entries recording last 10 branches, index by branch address
 - The pattern of the last 10 occurrences of that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

Comparing Predictors (Fig. 2.8)

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



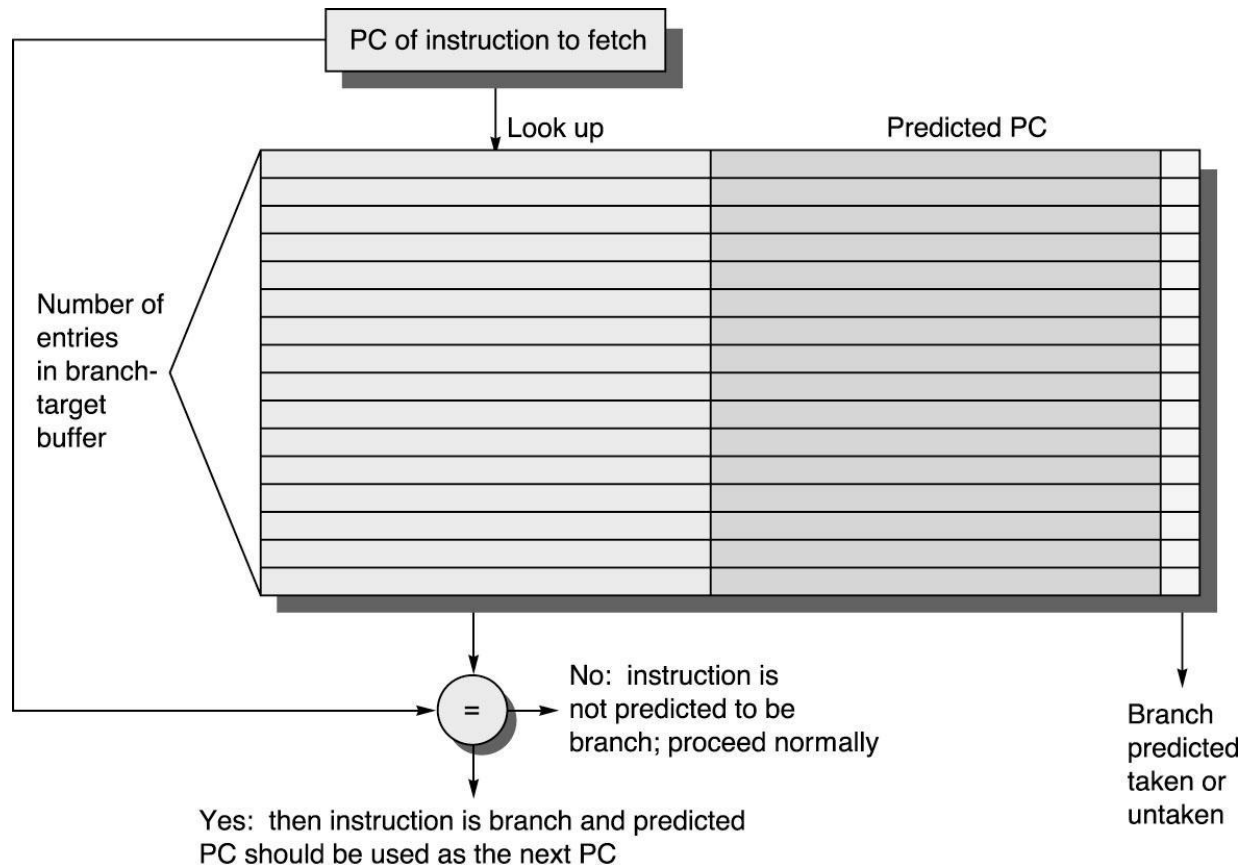
Pentium 4 Misprediction Rate (per 1000 instructions, not per branch)



Branch Target Buffers (BTB)

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

Branch Target Buffers



Dynamic Branch Prediction Summary

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

Outline

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

Advantages of Dynamic Scheduling

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

-
- we must separate the issue process into two parts:
 - checking for any structural hazards and waiting for the absence of a data hazard.
 - Thus we still use **in-order instruction issue** (i.e., instructions issued in program order
 - but we want an instruction to begin execution as soon as its data operands are available. Such a pipeline does **out-of-order execution**, which implies out-of-order completion.
 - Out-of-order execution introduces **the possibility of WAR and WAW hazards**, which do not exist in the five-stage integer pipeline and its logical extension to an in-order floating-point pipeline.

example

- `fdiv.d f0,f2,f4`
- `fmul.d f6,f0,f8`
- `fadd.d f0,f10,f14`

There is an **antidependence between the `fmul.d` and the `fadd.d`**

- if the pipeline executes the `fadd.d` before the `fmul.d` (which is waiting for the `fdiv.d`), it will violate the antidependence, yielding a WAR hazard.
- hazards are avoided by the use **of register renaming**

Dynamic Scheduling Step 1

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

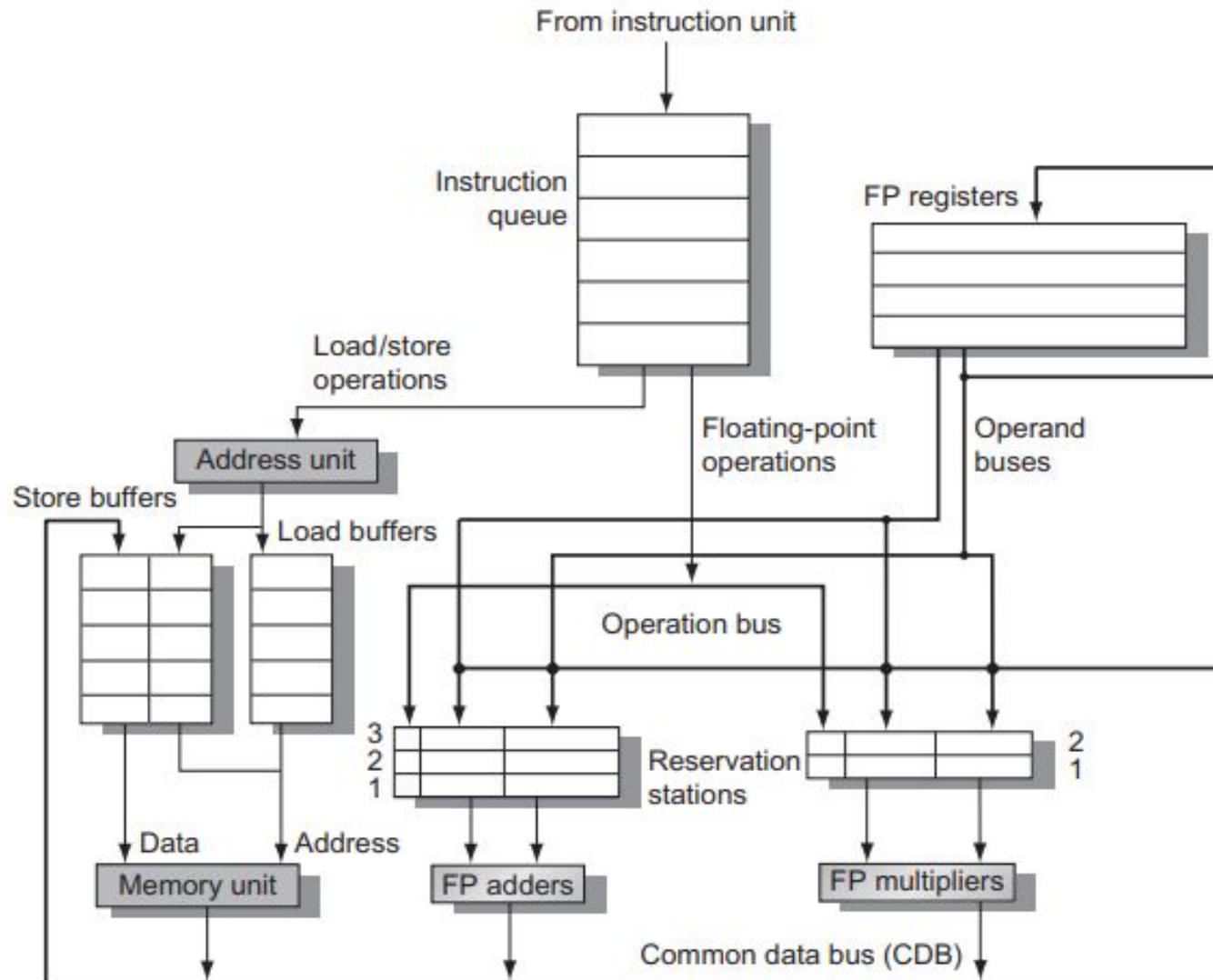
A Dynamic Algorithm: Tomasulo's

- **For IBM 360/91 (before caches!)**
 - \Rightarrow Long memory latency
- **Goal: High Performance without special compilers**
- **Small number of floating point registers (4 in 360) prevented interesting compiler scheduling of operations**
 - This led Tomasulo to try to figure out how to get more effective registers
 - **renaming in hardware!**

Tomasulo Algorithm

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

Tomasulo Organization



-
- Instructions are sent from the instruction unit into the instruction queue from which they are issued in **first-in, first-out (FIFO) order**.
 - The **reservation stations** include the operation and the actual operands, as well as information used for detecting and resolving hazards.
 - **Load buffers** have three functions:
 - (1) hold the components of the effective address until it is computed,
 - (2) track outstanding loads that are waiting on the memory, and
 - (3) hold the results of completed loads that are waiting for the CDB.

-
- Similarly, **store buffers** have three functions:
 - (1) hold the components of the effective address
 - until it is computed,
 - (2) hold the destination memory addresses of outstanding stores that are waiting for the data
 - value to store, and
 - (3) hold the address and value to store until the memory unit is available. All results from either the
 - **FP units or the load unit** are put on the CDB, which goes to the FP register file as well as to the reservation stations and store buffers.
 - The **FP adders** implement addition and subtraction, and
 - **the FP multipliers** do multiplication

1. Issue—

- Get the next instruction from the head of the instruction queue, which is maintained in FIFO order to ensure the maintenance of correct data flow.
- If there is a **matching reservation station that is empty**, issue the instruction to the station with the operand values,
- if they are currently in the registers. If **there is not an empty reservation station, then there is a structural hazard**, and the instruction issue stalls until a station or buffer is freed.
- If the operands are not in the registers, keep track of the functional units that will produce the operands.
- This step renames registers, eliminating WAR and WAW hazards.

2. Execute—

- If one or more of the operands is not yet available, monitor the common data bus while waiting for it to be computed.**
- When an operand becomes available, it is placed into any reservation station awaiting it.**
- When all the operands are available, the operation can be executed at the corresponding functional**
- unit.**
- By delaying instruction execution until the operands are available, RAW**
- hazards are avoided.**

-
- **3. Write result—**
 - **When the result is available, write it on the CDB and from there into the registers and into any reservation stations (including store buffers) waiting for this result.**
 - **Stores are buffered in the store buffer until both the value to be stored and the store address are available;**
 - **then the result is written as soon as the memory unit is free.**

Reservation Station Components

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

Vj, Vk: **Value** of Source operands

- Store buffers has V field, result to be stored

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

- Note: $Qj, Qk=0 \Rightarrow$ 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 /

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:

Reservation Stations:				$S1$	$S2$	RS	RS	
	$Time$	$Name$	$Busy$	Op	V_j	V_k	Q_j	Q_k
FU count down		Add1	No					
		Add2	No					
		Add3	No					
		Mult1	No					
		Mult2	No					

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

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>
FU									

0

Clock cycle
counter

Tomasulo Example Cycle 1

Instruction status:

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

Reservation Stations:

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

Register result status:

Clock												
	F0		F2		F4		F6		F8		F10	
1							Load1					

Tomasulo Example Cycle 2

Instruction status:

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

Reservation Stations:

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

Register result status:

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

Note: Can have multiple loads outstanding

Tomasulo Example Cycle 3

Instruction status:

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

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

Register result status:

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

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

Tomasulo Example Cycle 4

Instruction status:

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

Reservation Stations:

				S1	S2	RS	RS
Time	Name	Busy	Op	<i>I_j</i>	<i>I_k</i>	<i>Q_j</i>	<i>Q_k</i>
Add1		Yes	SUBD	M(A1)			Load2
Add2		No					
Add3		No					
Mult1		Yes	MULTD		R(F4)		Load2
Mult2		No					

Register result status:

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

- Load2 completing; what is waiting for Load2?

Tomasulo Example Cycle 5

Instruction status:

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

Reservation Stations:

				<i>S1</i>	<i>S2</i>	<i>RS</i>	<i>RS</i>
Time	Name	Busy	Op	<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:

				Exec		Write		
Instruction	<i>j</i>	<i>k</i>	Issue	Comp	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			Load3	No
SUBD	F8	F6	F2	4				
DIVD	F10	F0	F6	5				
ADDD	F6	F8	F2	6				

Reservation Stations:

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

Register result status:

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

- Issue ADDD here despite name dependency on F6?

Tomasulo Example Cycle 7

Instruction status:

				<i>Exec Write</i>			
Instruction	<i>j</i>	<i>k</i>	<i>Issue</i>	<i>Comp</i>	<i>Result</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		
DIVD	F10	F0	F6	5			
ADDD	F6	F8	F2	6			

Reservation Stations:

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

Register result status:

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

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

Tomasulo Example Cycle 8

Instruction status:

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

<i>Time</i>	<i>Name</i>	<i>Busy</i>	<i>Op</i>	<i>S1 S2 RS 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 F2 F4 F6 F8 F10 F12 ... F30</i>										
	<i>FU</i>										
8	Mult1	M(A2)		Add2	(M-M)	Mult2					

Tomasulo Example Cycle 9

Instruction status:

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

Reservation Stations:

				<i>S1</i>	<i>S2</i>	<i>RS</i>	<i>RS</i>
<i>Time</i>	<i>Name</i>	<i>Busy</i>	<i>Op</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:

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

Tomasulo Example Cycle 10

Instruction status:

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

Reservation Stations:

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

Register result status:

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

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

Tomasulo Example Cycle 11

Instruction status:

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

Reservation Stations:

				<i>S1</i>	<i>S2</i>	<i>RS</i>	<i>RS</i>
<i>Time</i>	<i>Name</i>	<i>Busy</i>	<i>Op</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:

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

<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					
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	FU								
	Mult1	M(A2)		(M-M+N	(M-M)	Mult2			

Tomasulo Example Cycle 13

Instruction status:

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

Reservation Stations:

				<i>S1</i>	<i>S2</i>	<i>RS</i>	<i>RS</i>
<i>Time</i>	<i>Name</i>	<i>Busy</i>	<i>Op</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	FU	Mult1	M(A2)		(M-M+N	(M-M)	Mult2			

Tomasulo Example Cycle 14

Instruction status:

Instruction		<i>j</i>	<i>k</i>	<i>Exec Write</i>			Busy	Address
				<i>Issue</i>	<i>Comp 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 S2 RS 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 F2 F4 F6 F8 F10 F12 ... F30</i>										
	FU Mult1 M(A2) (M-M+N (M-M) Mult2										

14

Tomasulo Example Cycle 15

Instruction status:

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

Reservation Stations:

				<i>S1</i>	<i>S2</i>	<i>RS</i>	<i>RS</i>
<i>Time</i>	<i>Name</i>	<i>Busy</i>	<i>Op</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		<i>F0</i>	<i>F2</i>	<i>F4</i>	<i>F6</i>	<i>F8</i>	<i>F10</i>	<i>F12</i>	...	<i>F30</i>
15	FU	Mult1	M(A2)		(M-M+N	(M-M)	Mult2			

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

Tomasulo Example Cycle 16

Instruction status:

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

				<i>S1</i>	<i>S2</i>	<i>RS</i>	<i>RS</i>
<i>Time</i>	<i>Name</i>	<i>Busy</i>	<i>Op</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

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

Tomasulo Example Cycle 55

Instruction status:

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

Reservation Stations:

				<i>S1</i>	<i>S2</i>	<i>RS</i>	<i>RS</i>
<i>Time</i>	<i>Name</i>	<i>Busy</i>	<i>Op</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	FU	M*F4	M(A2)		(M-M+N	(M-M)	Mult2			

Tomasulo Example Cycle 56

Instruction status:

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

Reservation Stations:

				<i>S1</i>	<i>S2</i>	<i>RS</i>	<i>RS</i>
<i>Time</i>	<i>Name</i>	<i>Busy</i>	<i>Op</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	FU	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:

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

				<i>S1</i>	<i>S2</i>	<i>RS</i>	<i>RS</i>
Time	Name	Busy	Op	<i>Vj</i>	<i>Vk</i>	<i>Qj</i>	<i>Qk</i>
	Add1	No					
	Add2	No					
	Add3	No					
	Mult1	No					
	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	FU	M*F4	M(A2)		(M-M+N	(M-M)	Result			

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

Why can Tomasulo overlap iterations of loops?

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

Tomasulo's scheme offers 2 major advantages

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

Tomasulo Drawbacks

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

And In Conclusion ... #1

- **Leverage Implicit Parallelism for Performance: Instruction Level Parallelism**
- **Loop unrolling by compiler to increase ILP**
- **Branch prediction to increase ILP**
- **Dynamic HW exploiting ILP**
 - Works when can't know dependence at compile time
 - Can hide L1 cache misses
 - Code for one machine runs well on another

And In Conclusion ... #2

- **Reservations stations: *renaming* to larger set of registers + buffering source operands**
 - Prevents registers as bottleneck
 - Avoids WAR, WAW hazards
 - Allows loop unrolling in HW
- **Not limited to basic blocks
(integer units gets ahead, beyond branches)**
- **Helps cache misses as well**
- **Lasting Contributions**
 - Dynamic scheduling
 - Register renaming
 - Load/store disambiguation
- **360/91 descendants are Intel Pentium 4, IBM Power 5, AMD Athlon/Opteron, ...**