

## Out-of-Order Execution

Several implementations

- **out-of-order completion**
  - \* IBM 360/91 with **Tomasulo's algorithm based on reservation stations**
  - CDC 6600 with **scoreboarding**
  - out-of-order completion leads to:
    - imprecise interrupts
- **in-order completion**
  - \* MIPS R10000/R12000 & Alpha 21264/21364 with **large physical register file & hardware register renaming**
  - Intel Pentium Pro/Pentium III with the **reorder buffer**

## Precise Interrupts

**Precise interrupts** preserve the model that instructions execute in program-generated order, one at a time

- if a recoverable interrupt occurs, the processor can recover from it
- instructions before the faulting instruction complete
- instructions after the faulting instruction can be restarted from scratch

What happens on a precise interrupt:

- **identify the instruction that caused the interrupt**
- **let the instructions before faulting instruction finish**
- disable writes for faulting instruction & subsequent instructions
- force trap instruction into pipeline
- trap routine
  - save the program state of the executing program
  - correct the cause of the interrupt
  - restore program state
- **restart faulting & subsequent instructions**

## Out-of-order Hardware

In order to compute correct results, need to keep track of:

- which instruction is in which stage of the pipeline
- which registers are being used for reading/writing & by which instructions
- which operands are available
- which instructions have executed
- which instructions have completed

Each scheme has different hardware structures & different algorithms to do this

## Tomasulo's Algorithm

**Tomasulo's Algorithm** (IBM 360/91)

- Introduced out-of-order execution capability plus hardware register renaming plus distributed execution plus snooping

### **Motivation**

- Improve floating point performance
  - only 4 FP registers
  - long FP delays
- wanted common compiler for all implementations

## Tomasulo's Algorithm

### Key features

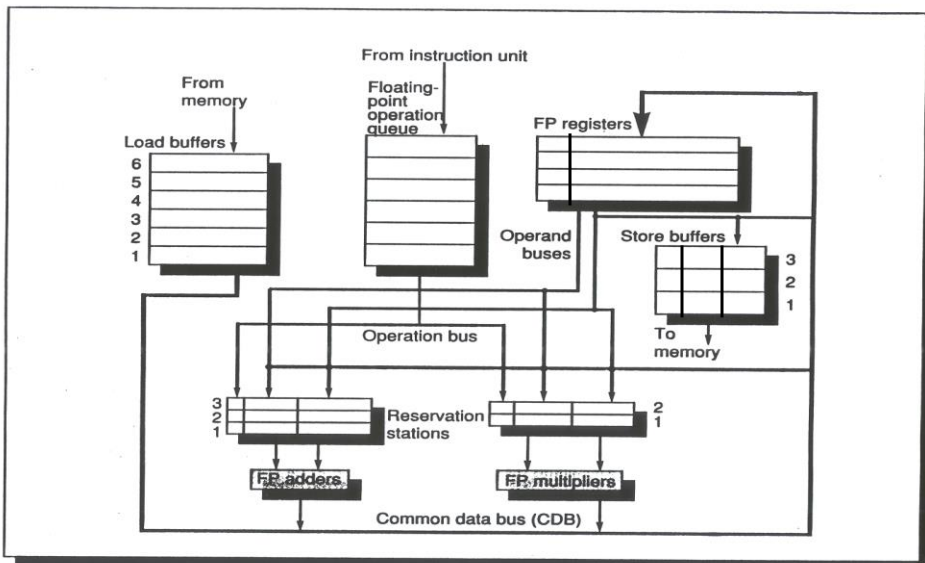
- distributed hazard detection & execution control
  - data hazard checks & forwarding to eliminate RAW hazards
  - hardware register renaming to eliminate WAR & WAW hazards
  - deciding which instruction to execute next
- multiple memories for storing data values: reservation stations
- common data bus
- dynamic memory disambiguation

Spring 2015

CSE 471: Out-of-Order Execution  
with Tomasulo's Algorithm

5

## Hardware for Tomasulo's Algorithm



Spring 2015

CSE 471: Out-of-Order Execution  
with Tomasulo's Algorithm

6

## Tomasulo's Algorithm : Key Features

### Reservation station

- buffer for a functional unit that holds instructions & their source operands
- source operand can be a **value** or the **name** of a reservation station entry or a load buffer entry that will provide the value
  - both operands don't have to be available at the same time
  - when both operand values are there, an instruction can be issued to its functional unit
- *preview*: eliminates WAR & WAW hazards

## Tomasulo's Algorithm: Key Features

### Common data bus (CDB)

- connects functional units & load buffer to reservations stations, registers, store buffer
- ships results to **all** hardware that could want an updated value at the same time
- *preview*: eliminates RAW hazards
  - not have to wait until registers are written before consuming a value

## Tomasulo's Algorithm: Key Features

### Distributed execution control

- each reservation station decides when to dispatch instructions to its function unit

### Distributed operand access

- **Tag** in each entry of the value consumer components (reservation stations, register file & store buffer) that indicates where its value will come from (the name)
- A producer component (functional units & load buffer) puts its computed value & a self-identifying **tag** on the common data bus
- Each consumer component entry monitors the CDB & grabs a value if its tag matches the tag on the CDB: **snooping**

## Tomasulo's Algorithm: Key Features

### Distributed Hazard Detection & Elimination

**RAW** hazards eliminated by **forwarding**

- source operand values that are produced after a consumer instruction's registers are read are tagged by the functional unit or load buffer entry that will produce them
- produced results are immediately forwarded to functional units on the common data bus
- don't have to wait until for value to be written into the register file

## Tomasulo's Algorithm: Key Features

### Distributed Hazard Detection & Elimination

Eliminate **WAR & WAW** hazards by (hardware) **register renaming**

- Tag of the operands in consumer hardware refer to the producing reservation station or load buffer entries for their values, not the registers
- Only the last instruction to write to a register updates it
- Reservation stations + registers, so eliminate more name dependences than a compiler & exploits more parallelism
- examples on next slide

## Tomasulo's Algorithm: Key Features

Recall that a **tag** in the reservation station/register file/store buffer entry indicates where the result will come from

### Handling WAW hazards

divf **F1**, F0, F8    **F1**'s tag *originally* specifies **divf**'s entry in the reservation station

...

subf **F1**, F8, F14    **F1**'s tag *now* specifies **subf**'s entry in the reservation station

no register will claim the **divf** result if it completes last

## Tomasulo's Algorithm: Key Features

### Handling WAR hazards

ld **F1**, \_

register **F1**'s tag *originally* specifies the entry in the load buffer for the **ld**

addf \_, **F1**, \_

**addf**'s reservation station entry specifies **ld**'s entry in the load buffer for source operand 1

...

subf **F1**, \_

register **F1**'s tag *now* specifies the reservation station entry that holds **subf**

Does not matter if **ld** finishes after **subf**; **F1** will no longer claim the load value & **addf** will use its tag (a load buffer entry) to get the loaded value

## Tomasulo's Algorithm: Key Features

### Dynamic memory disambiguation

- **the issue:** don't want loads to bypass stores to the same memory location
- **the solution:**
  - loads associatively check addresses in the store buffer
  - if an address matches, grab the value

## Tomasulo's Algorithm: Execution Steps

### **Tomasulo functions**

(assume the instruction has been fetched)

- **issue & read**
  - structural hazard detection for reservation station & load/store buffer entries
    - issue if no hazard
    - stall if hazard
  - read registers for source operands
    - put values into reservation stations if available
    - put tag of producing functional unit or load buffer entry if not  
(renaming the registers to eliminate WAR & WAW hazards)

## Tomasulo's Algorithm: Execution Steps

- **execute**
  - dispatch instruction from a reservation station to its functional unit if have both operand values
  - execute the operation
  - calculate effective address & start memory operation
- **write**
  - broadcast result & tag on the common data bus
  - reservation stations, registers & store buffer entries obtain the value through snooping (forwarding to eliminate RAW hazard)



## Tomasulo's Algorithm: State

**Tomasulo state:** the information that the hardware needs to control distributed execution

- **operation** of the issued instructions waiting for execution (Op)
  - located in reservation stations
- **tags** that indicate the producer for a source operand (Q)
  - located in reservation station entries, registers, store buffer entries (consumer hardware)
  - indicates what unit (reservation station or load buffer entry) will produce the operand
- **operand values** in reservation stations, registers & store buffers (V)
- reservation station & load/store buffer **busy fields** (Busy)
- **addresses** in load/store buffers (for memory disambiguation)

## Example in the Book: 1

**Instruction Status Table**

Instruction	Issue	Execute	Write Result	Which Cycle
ld F6, 34(R2)	yes	yes	yes	first load has executed
ld F2, 45(R3)	yes	yes		
multd F0, F2, F4	yes			
subd F8, F6, F2	yes			
divd F10, F0, F6	yes			
addd F6, F8, F2	yes			

**Reservation Stations**

Name	Busy	Op	V <sub>j</sub>	V <sub>k</sub>	Q <sub>j</sub>	Q <sub>k</sub>
Add1	yes	subd	(Load1)			Load2
Add2	yes	addd			Add1	Load2
Add3	no					
Mult1	yes	multd		(F4)	Load2	
Mult2	yes	divd		(Load1)	Mult1	

**Register Status (Q<sub>i</sub>)**

F0	F2	F4	F6	F8	F10	F12...
Mult1	Load2		Add2	Add1	Mult2	

## Example in the Book: 2

**Instruction Status Table**

Instruction	Issue	Execute	Write Result	Which Cycle
ld F6, 34(R2)	yes	yes	yes	second load has executed
ld F2, 45(R3)	yes	yes	yes	
multd F0, F2, F4	yes	yes		
subd F8, F6, F2	yes	yes		
divd F10, F0, F6	yes			
addd F6, F8, F2	yes			

**Reservation Stations**

Name	Busy	Op	V <sub>j</sub>	V <sub>k</sub>	Q <sub>j</sub>	Q <sub>k</sub>
Add1	yes	subd	(Load1)	(Load2)		
Add2	yes	addd		(Load2)	Add1	
Add3	no					
Mult1	yes	multd	(Load2)	(F4)		
Mult2	yes	divd		(Load1)	Mult1	

**Register Status (Q<sub>i</sub>)**

F0	F2	F4	F6	F8	F10	F12...
Mult1	0		Add2	Add1	Mult2	

Spring 2015

CSE 471: Out-of-Order Execution  
with Tomasulo's Algorithm

19

## Example in the Book: 3

**Instruction Status Table**

Instruction	Issue	Execute	Write Result	Which Cycle
ld F6, 34(R2)	yes	yes	yes	subtract has executed
ld F2, 45(R3)	yes	yes	yes	
multd F0, F2, F4	yes	yes		
subd F8, F6, F2	yes	yes	yes	
divd F10, F0, F6	yes			
addd F6, F8, F2	yes	yes		

**Reservation Stations**

Name	Busy	Op	V <sub>j</sub>	V <sub>k</sub>	Q <sub>j</sub>	Q <sub>k</sub>
Add1	no					
Add2	yes	addd	(add1)	(Load2)		
Add3	no					
Mult1	yes	multd	(Load2)	(F4)		
Mult2	yes	divd		(Load1)	Mult1	

**Register Status (Q<sub>i</sub>)**

F0	F2	F4	F6	F8	F10	F12...
Mult1	0		Add2	0	Mult2	

Spring 2015

CSE 471: Out-of-Order Execution  
with Tomasulo's Algorithm

20

### Example in the Book: 4

**Instruction Status Table**

Instruction	Issue	Execute	Write Result	Which Cycle
ld F6, 34(R2)	yes	yes	yes	<b>add has executed</b>
ld F2, 45(R3)	yes	yes	yes	
multd F0, F2, F4	yes	yes		
subd F8, F6, F2	yes	yes	yes	
divd F10, F0, F6	yes			
addd F6, F8, F2	yes	yes	yes	

**Reservation Stations**

Name	Busy	Op	V <sub>j</sub>	V <sub>k</sub>	Q <sub>j</sub>	Q <sub>k</sub>
Add1	no					
Add2	no					
Add3	no					
Mult1	yes	multd	(Load2)	(F4)		
Mult2	yes	divd		(Load1)	Mult1	

**Register Status (Q<sub>i</sub>)**

F0	F2	F4	F6	F8	F10	F12...
Mult1	0		0	0	Mult2	

Spring 2015

CSE 471: Out-of-Order Execution  
with Tomasulo's Algorithm

21

### Example in the Book: 5

**Instruction Status Table**

Instruction	Issue	Execute	Write Result	Which Cycle
ld F6, 34(R2)	yes	yes	yes	<b>multiply has executed</b>
ld F2, 45(R3)	yes	yes	yes	
multd F0, F2, F4	yes	yes	yes	
subd F8, F6, F2	yes	yes	yes	
divd F10, F0, F6	yes	yes		
addd F6, F8, F2	yes	yes	yes	

**Reservation Stations**

Name	Busy	Op	V <sub>j</sub>	V <sub>k</sub>	Q <sub>j</sub>	Q <sub>k</sub>
Add1	no					
Add2	no					
Add3	no					
Mult1	no					
Mult2	yes	divd	(mult1)	(Load1)		

**Register Status (Q<sub>i</sub>)**

F0	F2	F4	F6	F8	F10	F12...
0	0		0	0	Mult2	

Spring 2015

CSE 471: Out-of-Order Execution  
with Tomasulo's Algorithm

22

## Tomasulo's Algorithm

### Dynamic loop unrolling

```
LOOP: ld F0, 0(R1)
      addf F0, F0, F1
      st F0, 0(R1)
      sub R1, R1, #8
      bnez R1, LOOP
```

- addf and st in each iteration has a different tag for the F0 value
- only the store in the last iteration writes to F0
- effectively completely unrolling the loop

## Tomasulo's Algorithm

### Dynamic loop unrolling

Nice features relative to static loop unrolling

- effectively increases number of registers (reservations station entries, load buffer entries, registers) but without register pressure
- simpler (1960) compiler

Downside

- loop control instructions still executed

## Dynamic Scheduling

### Advantages over static scheduling

- more places to hold register values
- makes dispatch decisions dynamically, based on when instructions actually complete & operands are available
- can *completely* disambiguate memory references

### Effects of these advantages

- ⇒ more effective at exploiting instruction-level parallelism (especially given compiler technology at the time, but still true now)
  - increased instruction throughput
  - increased functional unit utilization
- ⇒ efficient execution of code compiled for a different pipeline

### Use both!