Out-of-Order Execution

Several implementations

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

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 completed

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

Tomasulo's Algorithm

Tomasulo's Algorithm (IBM 360/91)

- out-of-order execution capability plus register renaming

Motivation

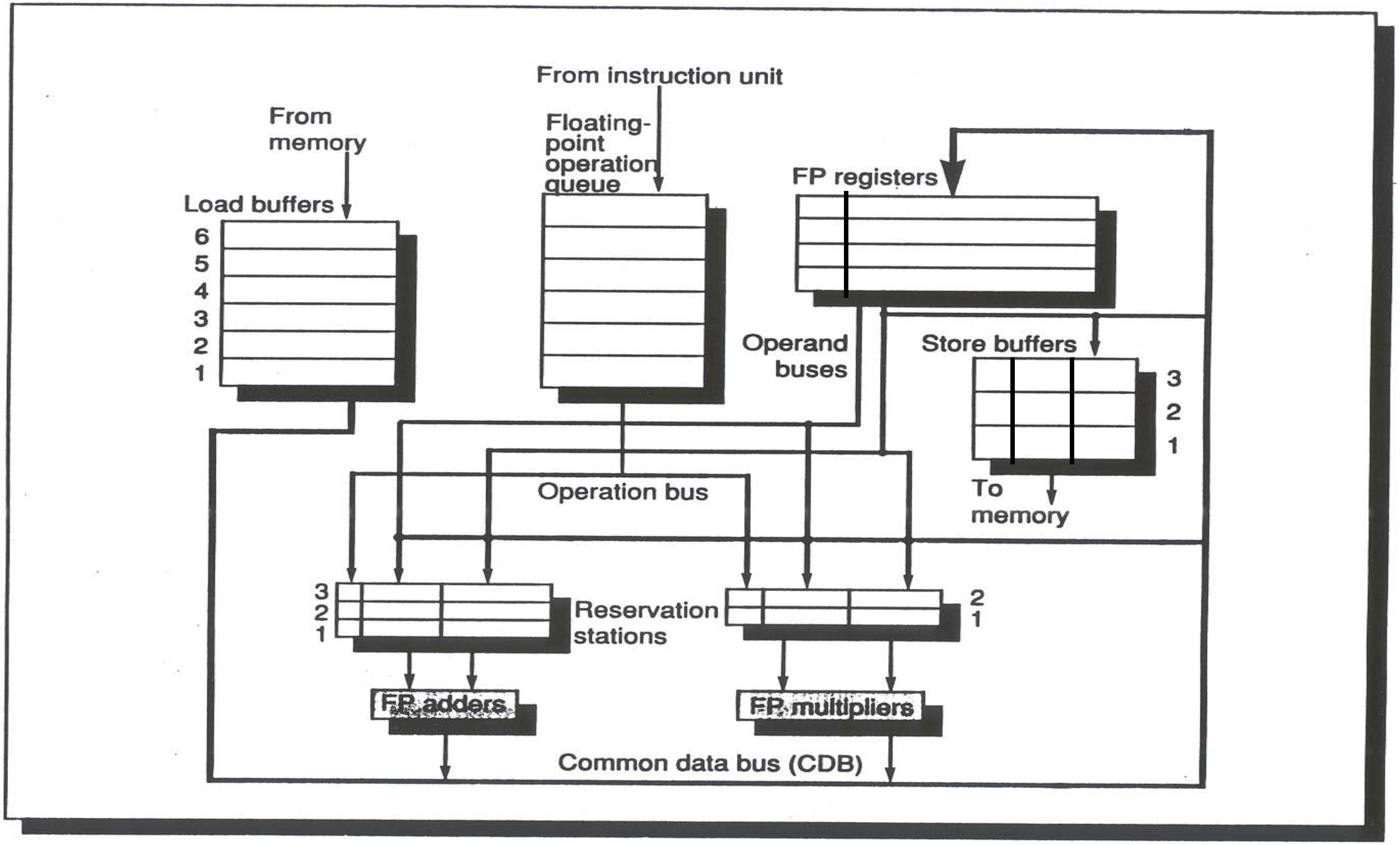
- only 4 FP registers
- long FP delays
- wanted common compiler for all implementations

Tomasulo's Algorithm

Key features & hardware structures

- reservation stations
- distributed hazard detection & execution control
  - forwarding to eliminate RAW hazards
  - register renaming to eliminate WAR & WAW hazards
  - deciding which instruction to execute next
- common data bus
- dynamic memory disambiguation

Hardware for Tomasulo's Algorithm



Tomasulo’s Algorithm: Key Features

Reservation stations

- buffers for functional units that hold instructions stalled for RAW hazards & their operands
- source operands can be ***values*** or ***names of other reservation station entries*** or ***load buffer entries*** that will produce the value
    - both operands don’t have to be available at the same time
    - when both operand values have been computed, an instruction can be dispatched to its functional unit

Tomasulo’s Algorithm: Key Features

Eliminating Hazards

RAW hazards eliminated by **forwarding**

- source operand values that are computed after the registers are read are known by the functional unit or load buffer that will produce them
- 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

Eliminating Hazards

Eliminate WAR & WAW hazards by **register renaming**

- name-dependent instructions refer to reservation station or load buffer locations for their sources, not the registers
- the last writer to the register updates it
- more reservation stations than registers, so eliminates more name dependences than a compiler can & exploits more parallelism
- examples on next slide

Tomasulo's Algorithm: Key Features

Register renaming eliminates WAR & WAW hazards

- **Tag** in the reservation station/register file/store buffer 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 registration station entry that holds `subf`

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

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
- eliminates RAW hazards: not have to wait until registers are written before consuming a value

Tomasulo’s Algorithm: Key Features

Distributed hazard detection & execution control

- each reservation station decides when to dispatch instructions to its function unit
- each hardware data structure entry that needs values grabs the values itself: **snooping**
  - reservation stations, store buffer entries & registers have a tag saying where their data should come from
  - result producer puts its value & a self-identifying tag on the common data bus
  - when the tags match, reservation stations, store buffer entries & registers grab the data

Tomasulo’s Algorithm: Key Features

Dynamic memory disambiguation

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

Tomasulo’s Algorithm: Execution Steps

Tomasulo functions

(assume the instruction has been fetched)

- **issue & read**
  - structural hazard detection for reservation stations & load/store buffers
    - issue if no hazard
    - stall if hazard
  - read registers for source operands
    - put into reservation stations if values are in them
    - put tag of producing functional unit or load buffer if not

      (renaming the registers to eliminate WAR & WAW hazards)

Tomasulo’s Algorithm: Execution Steps

- **execute**
  - RAW hazard detection
  - snoop on common data bus for missing operands
  - dispatch instruction to a functional unit when obtain both operand values
  - execute the operation
  - calculate effective address & start memory operation
- **write**
  - broadcast result & reservation station id (tag) on the common data bus
  - reservation stations, registers & store buffer entries obtain the value through snooping

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 stations, registers, store buffer entries
    - what unit (reservation station or load buffer) will produce the operand
    - special value (blank for us) if value already there
- **operand values** in reservation stations & load/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_j	V_k	Q_j	Q_k
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_i)

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_j	V_k	Q_j	Q_k
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_i)

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

Example in the Book: 3

Instruction Status Table

Instruction	Issue	Execute	Write Result	Which Cycle
`ld F6, 34(R2)`	yes	yes	yes	
`ld F2, 45(R3)`	yes	yes	yes	subtract has executed
`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_j	V_k	Q_j	Q_k
Add1	no					
Add2	yes	addd	(add1)	(Load2)		
Add3	no					
Mult1	yes	multd	(Load2)	(F4)		
Mult2	yes	divd		(Load1)	Mult1	

Register Status (Q_i)

F0	F2	F4	F6	F8	F10	F12...
Mult1	()		Add2	()	Mult2	

Example in the Book: 4

Instruction Status Table

Instruction	Issue	Execute	Write Result	Which Cycle
`ld F6, 34(R2)`	yes	yes	yes	add 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	yes	

Reservation Stations

Name	Busy	Op	V_j	V_k	Q_j	Q_k
Add1	no					
Add2	no					
Add3	no					
Mult1	yes	multd	(Load2)	(F4)		
Mult2	yes	divd		(Load1)	Mult1	

Register Status (Q_i)

F0	F2	F4	F6	F8	F10	F12...
Mult1	()		()	()	Mult2	

Example in the Book: 5

Instruction Status Table

Instruction	Issue	Execute	Write Result	Which Cycle
`ld F6, 34(R2)`	yes	yes	yes	multiply has executed
`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_j	V_k	Q_j	Q_k
Add1	no					
Add2	no					
Add3	no					
Mult1	no					
Mult2	yes	divd	(mult1)	(Load1)		

Register Status (Q_i)

F0	F2	F4	F6	F8	F10	F12...
()	()		()	()	Mult2	

Tomasulo's Algorithm

Dynamic loop unrolling

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

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

Tomasulo’s Algorithm

Dynamic loop unrolling

Nice features relative to static loop unrolling

- effectively increases number of registers (# reservations stations, load buffer entries, registers) but without register pressure
- dynamic memory disambiguation to prevent loads after stores with the same address from getting old data if they execute first
- simpler (1960) compiler

Downside

- loop control instructions still executed
- much more complex hardware

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 parallelism (especially given compiler technology at the time)

- increased instruction throughput
- increased functional unit utilization

⇒ efficient execution of code compiled for a different pipeline

⇒ simpler compiler in theory

Use both!