# 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
- when instructions have completed

# Out-of-Order Execution

Several implementations

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

# Tomasulo's Algorithm

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

- out-or-order execution capability plus register renaming

**Motivation**

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

**Key features**

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

# Tomasulo's Algorithm: Key Features

**Reservation stations**: buffers for functional units that hold instructions & operands stalled for RAW hazards

- source operands can be *values* or *names of other reservation stations* that will produce the value
  - both operands don't have to be available at the same time
  - when both operand values have been computed, instruction can be dispatched to its functional unit
- RAW hazards eliminated for **forwarding**
  - results forwarded to functional units & are available immediately
  - forwarding done on the common data bus (later slide)
  - don't have to read the registers
- **register renaming** to eliminate WAR & WAW hazards
  - name dependent instructions refer to reservation station locations for their sources, not the registers (as above)
  - last writer to the register updates it
  - more reservation stations than registers, so eliminates more name dependences than a compiler can
  - examples on next slide

# Handling WAR & WAW Hazards

Register renaming eliminates WAR & WAW hazards

- **Tag** in the reservation station/register file/store buffer indicates where the result will come from

## Handling WAR hazards

| | |
|---|---|
| `ld F1,_` | F1's tag *originally* specifies the **`ld`** entry in the load buffer |
| `addf _, F1,_` | `addf`'s reservation station entry specifies the **`ld`** entry in the load buffer for source operand 1 |
| `addf F1,_` | F1's tag *now* specifies the **`addf`** reservation station |

not matter if `ld` finishes after `addf`

## Handling WAW hazards

| | |
|---|---|
| `addf F1,F0,F8` | F1's tag *originally* specifies **`addf`**'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 `addf` result if it completes last

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

## Distributed hazard detection & execution control

- each reservation station decides which instructions to dispatch to function units & when
- each hardware data structure that needs values grabs the values itself: **snooping**
  - reservation stations & registers have a tag saying where their data should come from
  - when it matches the tag value on the bus, reservation stations & registers grab the data

## 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
      - this is the register renaming to eliminate WAR & WAW hazards
- **execute**
  - RAW hazard detection
  - snoop on common data bus for missing operands
  - dispatch to functional unit when obtain both operand values
- **write**
  - broadcast result & reservation station id (tag) on the common data bus to reservation stations, registers & store buffer

# 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)
  - reservation stations
- tags that indicate the producer for a source operand (Q)
  - what unit (reservation station or load buffer) will produce the operand
  - 0 or blank if value already there
  - reservation stations, registers, store buffer
- 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 | Issue | Execute | Write Result | Which Cycle |
|---|---|---|---|---|
| `ld F6, 34(R2)` | yes | yes | **yes** | **cycle after first load has com-pleted** |
| `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 | sub | **(Load1)** | | | Load2 |
| Add2 | yes | add | | | Add1 | Load2 |
| Add3 | no | | | | | |
| Mult1 | yes | mult | | (F4) | Load2 | |
| Mult2 | yes | div | | **(Load1)** | Mult1 | |

**Register Status ($Q_i$)**

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

# Example in the Book 2

| Instruction | Issue | Execute | Write Result | Which Cycle |
|---|---|---|---|---|
| `ld F6, 34(R2)` | yes | yes | yes | cycle after second load has completed |
| `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 | sub | (Load1) | (Load2) | | |
| Add2 | yes | add | | (Load2) | Add1 | |
| Add3 | no | | | | | |
| Mult1 | yes | mult | (Load2) | (F4) | | |
| Mult2 | yes | div | | (Load1) | Mult1 | |

**Register Status ($Q_i$)**

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

# Example in the Book 3

| Instruction | Issue | Execute | Write Result | Which Cycle |
|---|---|---|---|---|
| `ld F6, 34(R2)` | yes | yes | yes | cycle after subtract has com-pleted |
| `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 | $V_j$ | $V_k$ | $Q_j$ | $Q_k$ |
|---|---|---|---|---|---|---|
| Add1 | no | sub | | | | |
| Add2 | yes | add | (add1) | (Load2) | | |
| Add3 | no | | | | | |
| Mult1 | yes | mult | (Load2) | (F4) | | |
| Mult2 | yes | div | | (Load1) | Mult1 | |

**Register Status ($Q_i$)**

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

# Example in the Book 4

| Instruction | Issue | Execute | Write Result | Which Cycle |
|---|---|---|---|---|
| `ld F6, 34(R2)` | yes | yes | yes | cycle after add has completed |
| `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 | mult | (Load2) | (F4) | | |
| Mult2 | yes | div | | (Load1) | Mult1 | |

**Register Status ($Q_i$)**

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

# Example in the Book 5

| Instruction | Issue | Execute | Write Result | Which Cycle |
|---|---|---|---|---|
| `ld F6, 34(R2)` | yes | yes | yes | cycle after mult has completed |
| `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 | div | (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

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

Nice features relative to static loop unrolling

- effectively increases number of registers (# reservations stations, load buffer entries, registers) – less register pressure
- dynamic memory disambiguation to prevent loads after stores with the same address from getting old data if they execute first
- simpler compiler

Downside

- loop control instructions still executed
- much more complex hardware

# Dynamic Scheduling

## Advantages over static scheduling

- more registers to work with
- makes dispatch decisions dynamically, based on when instructions actually complete & operands are available
- can *completely* disambiguate memory references

  $\Rightarrow$ more effective at exploiting parallelism 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!