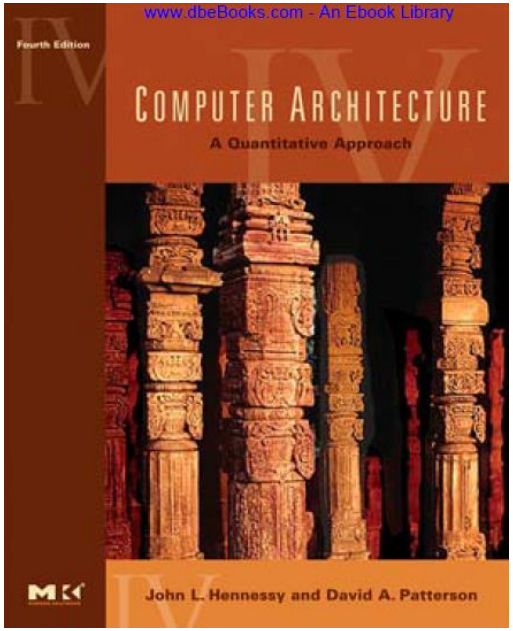
Instruction-Level Parallelism

dynamic scheduling

prepared and instructed by
Shmuel Wimer
Eng. Faculty, Bar-Ilan University



Dynamic Scheduling

Reorder instruction execution to reduce stalls while maintaining data flow and exception behavior.

- Enables handling some cases when dependences are unknown at compile time (e.g. memory reference).
- Simplifies the compiler.
- Allows the processor to tolerate cache misses delays by executing other code while waiting for miss resolution.
- Allows code compiled for one pipeline to run efficiently on a different pipeline.
- Increases significantly the hardware complexity.

Ordinary pipeline issues and executes instructions **in-line**.

- If instruction ***j*** depends on instruction ***i***, all instructions after ***j*** must be stalled until ***i*** is finished and ***j*** can execute.

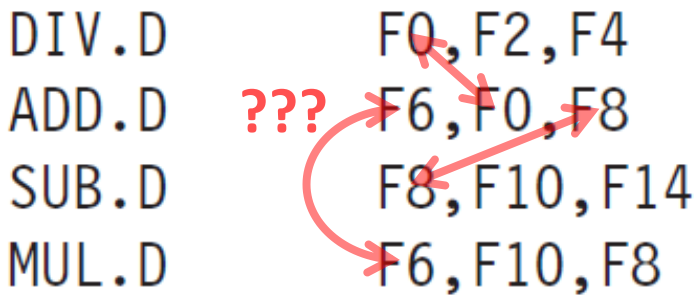
```
DIV.D       F0,F2,F4
ADD.D       F10,F0,F8
SUB.D       F12,F8,F14
```

SUB.D is independent, but is not executed because ADD.D dependence on DIV.D stalls the pipeline.

Out-Of-Order Execution

Instructions still issued **in-order**, but start execution as soon as their operands are available **out-of-order** (**OOO**), implying **out-of-order completion**.

OOO introduces possibilities of **WAR** and **WAW** hazards, not existing in in-order pipeline.



Executing SUB.D before ADD.D (waits for F0) causes a **WAR** hazard.

WAW also occurs if F6 is written by MUL.D before ADD.D.

Register renaming avoids these hazards.

OOO completion must preserve **exception behavior** to happen **exactly** as by in-order.

- No instruction should raise an exception until the CPU knows that it will be surely executed.

OOO splits the ID into two stages:

1. **Issue**—Decodes instructions, checks for **structural hazards**.
2. **Read operands**—Waits until no **data hazards**, then reads operands.

IF precedes **issue**, fetching into an instructions queue, from where Instructions are issued.

EX follows **read operands** and may take multiple cycles, depending on the operation.

Pipeline should support simultaneous execution of multiple instructions.

- Without it a major advantage of **OOO** is lost.

Instructions are issued **in-order**, but can enter execution **OOO**.

OOO HW implements **Tomasulo's Algorithm.**

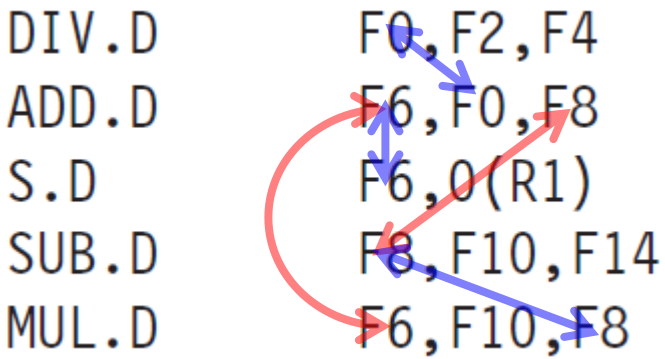
Tomasulo's Dynamic Scheduling

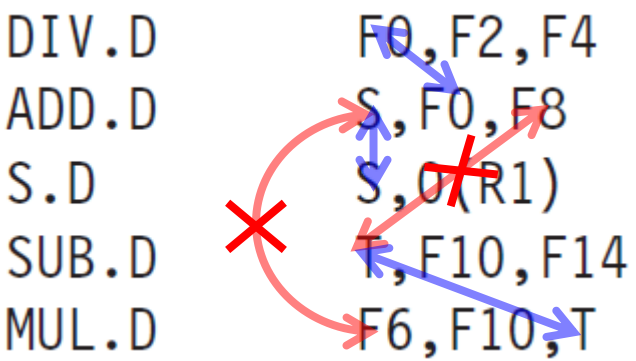
Invented for IBM 360/91 FPU by Robert Tomasulo.

- Eliminates **RAW** hazards by tracking operands readiness.
- Eliminates **WAR** and **WAW** hazards by register renaming.

Assume **FPU** and **load-store** unit, use MIPS ISA.

- **Rename** all destination registers, including those with pending read and write for earlier instructions.
- **OOO** writes do not affect instructions depending on earlier value of an operand.





WAR hazard, and **WAW** hazard if MUL.D finishes before ADD.D (called **name dependence**).

True data dependencies (**RAW** hazards).

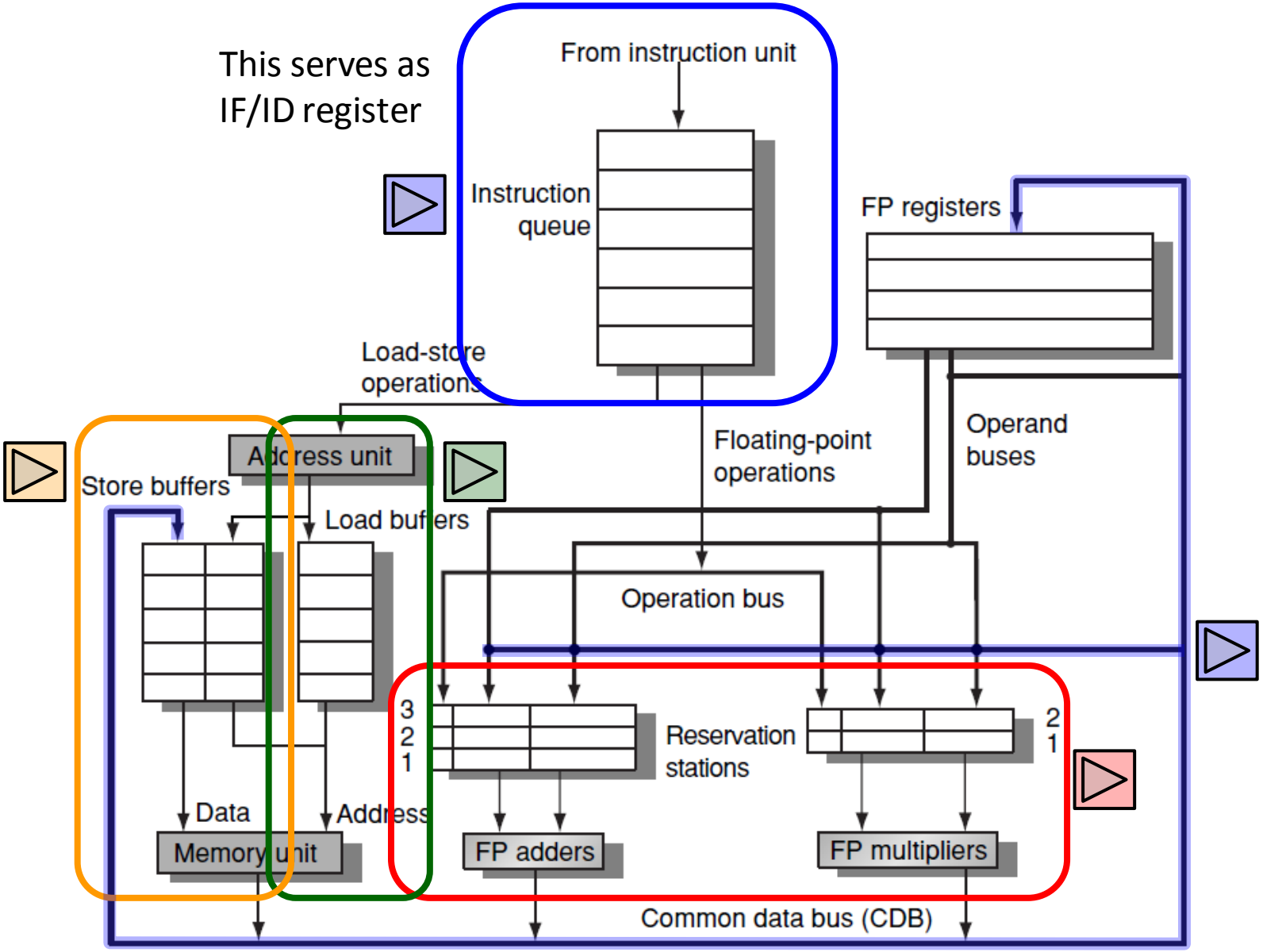
WAR and **WAW** are eliminated by **register renaming**.

Subsequent usage of **F8** must be replaced by **T**.

Very difficult for compiler (branches may intervene). Tomasulo's can.

Hardware Implementation

This serves as IF/ID register



Instructions are sent from the instruction unit into a **queue** from where they issue in FIFO order.

RSs include the **operations** and the actual **operands**, together with information for hazard detection and resolution.

Load buffers:

1. hold the components of the effective address until it is computed,
2. track outstanding loads waiting on memory, and
3. hold the results of completed loads, waiting for the CDB.

Store buffers:

1. hold the components of the effective address until it is computed,
2. hold the destination addresses of outstanding stores waiting for the data value to store, and
3. hold the address and data to store until the memory unit is available.

All results of the FPU and load unit are put on the **CDB**, which goes to the FP registers, to the RSs and to the store buffers.

The adder implements also subtraction and the multiplier implements also division.

Reservation Station (**RS**) implements the renaming, buffering the operands of instructions waiting to issue.

RS fetches and buffers an operand as soon as it is ready, eliminating the need to get it from the **Reg. File** (**RF**).

Pending instruction designate the **RS** that will provide their operands.

pending operands are renamed at **issue** from **RF** specifier to **RS**.

When successive writes to RF (**WAW**) overlap in execution, only the latest issued updates the RF.

There are more RSs than real registers, so it can eliminate hazards that compiler could not.

In ordinary CPU hazard detection and execution control was **centralized**. Here it is **distributed**.

The info at each **RS** of a functional unit (**FU**) determines when an instruction can start execution at that unit.

RS passes results directly to the **FU**s where the results are required through **Common Data Bus** (**CDB**) rather than going through **RF**.

Pipeline issuing **multiple** instructions and supporting **multiple** execution units requires more than one **CDB**.

The Steps of an Instruction

1. Issue

Get next instruction from the head of the queue(FIFO) and hence issued in-order.

If an empty matched **RS** is available, issue the instruction to that **RS** together with the operands if they are currently exist in RF.

If not in RF, keep track of **FU** producing them. This steps **renames registers**, eliminating **WAR** and **WAW** hazards.

If a matched **RS** is not available, there is a **structural hazard**. Instruction in queue **stalls** until **RS** is freed.

2. Execute

FP, Int: If an operand is not yet available, monitor **CDB** for its readiness.

When available, it is placed at any **RS** awaiting it. When all operation's operands are available it is executed.

By delaying operations until all their operands are available **RAW** hazards are avoided.

Several instructions could become ready on the same CLK cycle.

Independent units can start execution in same cycle.

If few instructions are ready for the same **FPU**, choice can be arbitrary.

Load and stores: Require **two-step execution** process.

The 1st step computes the **effective address** when the register is available. The address is placed in the load or store buffer.

Load is executed as soon as memory unit is available.

Store waits for the **value** before sent to memory unit.

Load and **stores** are maintained in the program order to prevent **hazards through memory**.

To preserve **exception** behavior, instructions do not initiate execution until all preceding branches in program order have completed.

The CPU must know that the **BP** is correct before allowing execution of instruction after **BP** (in program).

This guarantees that only instructions that would really be executed raise an exception.

Speculation will provide more complete solution

3. Write Results

When the result is available, put it on the **CDB** and from there into the **RF** and any **RS** waiting for the result.

Stores are buffered into the **store buffer** until both the value to be stored and the store address are available.

The result is written from **store buffer** as soon as memory unit is free.

The Reservation Station Data Structure

Each **RS** has seven fields:

- **Op** – The operation to perform on the source operands S1 and S2.
- **Qj** , **Qk** – The **RS** that will produce S1 and S2. Qj = 0 or Qk = 0 indicates that the source operands are available in Vj or Vk , or operand is unnecessary.
- **Vj** , **Vk** – The values of S1 and S2.
- **A** – Holds information for the memory address calculation for load or store.
- **Busy** – This RS and its functional unit are occupied.

Each **RF** register has the field:

- **Qi** – # of RS containing the operation whose result should be stored into the register.

  Qi=0 if no active instruction is computing a result destined for this register. Register contents is valid.

Load and store buffers have field A, holding the effective address once the first execution step (of two) is completed.

Example: What is Tomasulo's DS when the first load has completed and written its result?

```
L.D     F6,32(R2)
L.D     F2,44(R3)
MUL.D   F0,F2,F4
SUB.D   F8,F2,F6
DIV.D   F10,F0,F6
ADD.D   F6,F8,F2
```

Instruction status

Instruction		Issue	Execute	Write Result
L.D	F6,32(R2)	√	√	√
L.D	F2,44(R3)	√	√	
MUL.D	F0,F2,F4	√		
SUB.D	F8,F2,F6	√		
DIV.D	F10,F0,F6	√		
ADD.D	F6,F8,F2	√		

Instruction status is not a part of the hardware

Reservation station

Name	Busy	Op	Vj	Vk	Qj	Qk	A
Load1	no						
Load2	yes	Load					44 + Regs[R3]
Add1	yes	SUB		Mem[32 + Regs[R2]]	Load2		
Add2	yes	ADD			Add1	Load2	
Add3	no						
Mult1	yes	MUL		Regs[F4]	Load2		
Mult2	yes	DIV		Mem[32 + Regs[R2]]	Mult1		

```
L.D     F6,32(R2)
L.D     F2,44(R3)
MUL.D   F0,F2,F4
SUB.D   F8,F2,F6
DIV.D   F10,F0,F6
ADD.D   F6,F8,F2
```

Register status

F0	F2	F4	F6	F8	F10
Mult1	Load2		Add2	Add1	Mult2

WAR hazard involving R6 is eliminated in one of two ways.

```
L.D     F6,32(R2)
L.D     F2,44(R3)
MUL.D   F0,F2,F4
SUB.D   F8,F2,F6
DIV.D   F10,F0,F6
ADD.D   F6,F8,F2
```

If L.D completed, Vk of DIV.D stores the result and is independent of ADD.D (as shown in instruction status).

If L.D not completed, Qk of DIV.D points to Load1 RS and DIV.D would be independent of ADD.D.

In either case ADD.D can issue and execute without affecting DIV.D.

Homework: study the example below.

Example: Assume the following latencies: load 1 cycle, add 2 cycles, multiply 6 cycles and divide 12 cycles.

What the status tables look like when the MUL.D is ready to write result?

Latency	Instruction status Instruction		Issue	Execute	Write Result
1	L.D	F6,32(R2)	√	√	√
1	L.D	F2,44(R3)	√	√	√
6	MUL.D	F0,F2,F4	√	√	
2	SUB.D	F8,F2,F6	√	√	√
12	DIV.D	F10,F0,F6	√		
2	ADD.D	F6,F8,F2	√	√	√

Reservation station

Name	Busy	Op	Vj	Vk	Qj	Qk	A
Load1	no						
Load2	no						
Add1	no		Load1				
Add2	no						
Add3	no						
Mult1	yes	MUL	Mem[44+Regs[R3]]	Regs[F4]			
Mult2	yes	DIV		Load1	Mult1		

```
L.D     F6,32(R2)
L.D     F2,44(R3)
MUL.D   F0,F2,F4
SUB.D   F8,F2,F6
DIV.D   F10,F0,F6
ADD.D   F6,F8,F2
```

Register status

Field	F0	F2	F4	F6	F8	F10
Qi	Mult1					Mult2

Add has been completed since the operands of DIV.D were copied, thereby avoiding the WAR hazard in F6.

Even if the **load of F6** was delayed, the add into F6 could be executed without triggering a WAW hazard.

Tomasulo Algorithm Details

Homework: study the details below.

Instruction state	Wait until	Action or bookkeeping

Issue FP operation	Station **r** empty	`if (RegisterStat[rs].Qi≠0)`
		`  {RS[r].Qj ← RegisterStat[rs].Qi}`
		`else {RS[r].Vj ← Regs[rs]; RS[r].Qj ← 0};`
		`if (RegisterStat[rt].Qi≠0)`
		`  {RS[r].Qk ← RegisterStat[rt].Qi`
		`else {RS[r].Vk ← Regs[rt]; RS[r].Qk ← 0};`
		`RS[r].Busy ← yes; RegisterStat[rd].Qi← r;`

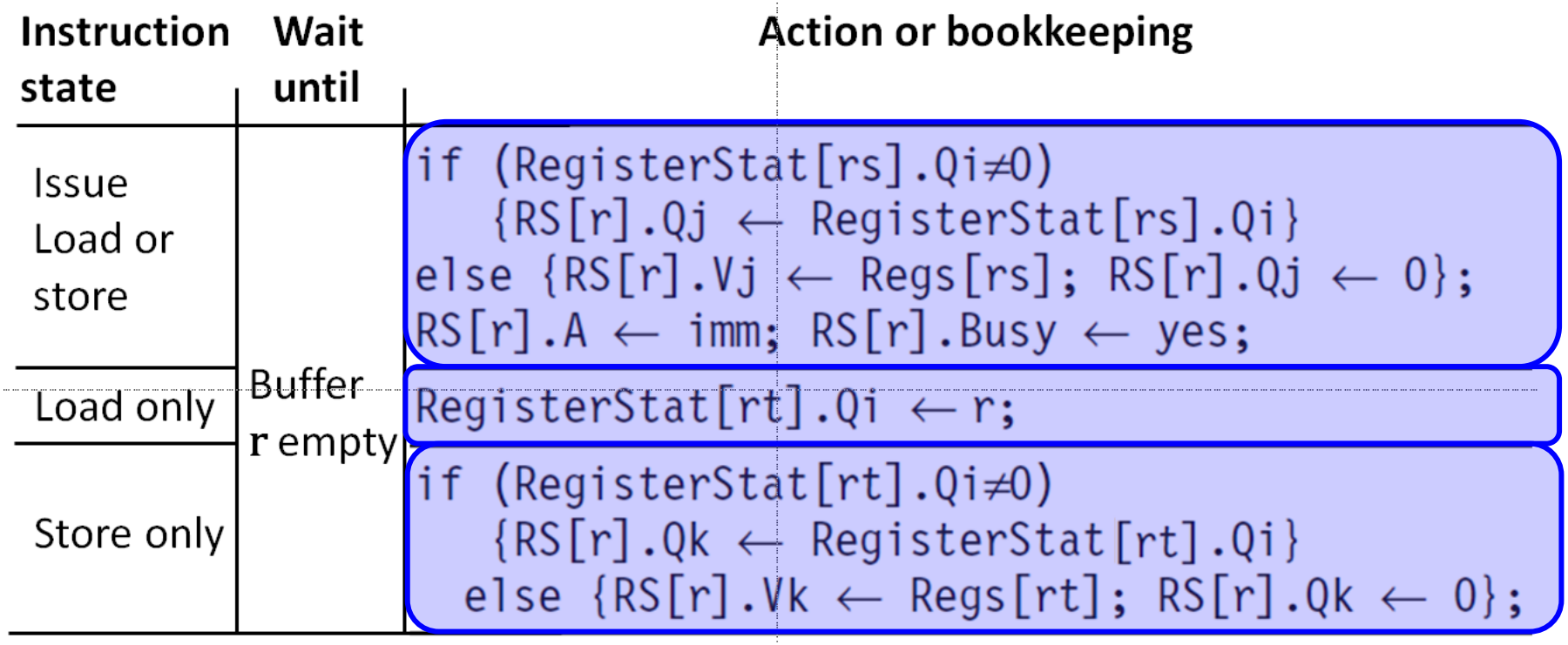
rs and **rt** are the source registers. **rd** is the destination register. **r** is the reservation station (**RS**) or buffer that the instruction is assigned to. **Regs**[·] is the register file, **RegisterStat**[·] is the register status.

If the operands are available in the registers, they are stored in the V fields. Otherwise, the Q fields are set to indicate the RS that will produce the values needed as source operands.

The instruction waits at the RS until both its operands are available, indicated by zero in the Q fields.

The Q fields are set to zero either when this instruction is issued, or when an instruction on which this instruction depends completes and does its write back.

When an instruction has finished execution and the CDB is available, it can do its write back.



imm is the sign-extended immediate field.

Instruction state	Wait until	Action or bookkeeping
Execute FP operation	RS[r].Qj=0 and RS[r].Qk=0	Compute results. Operands are in Vj and Vk ;
Execute Load-store step 1	RS[r].Qj=0 and r is head of load-store queue	RS[r].A ← RS[r].Vj+RS[r].A ;
Execute Load step 2	Load step 1 complete	Read from Mem[RS[r].A] ;

All the buffers, registers, and RSs whose value of Qj or Qk is the same as the completing RS, update their values from the CDB and mark their Q fields with zero to indicate that values have been received.

Instruction state	Wait until	Action or bookkeeping
Write result of FP operation or load	Execution complete at r and CDB available	∀x (if (RegisterStat[x].Qi=r) {Regs[x]←result ; RegisterStat[x].Qi←0 }) ; ∀x (if (RS[x].Qj=r) {RS[x].Vj←result ; RS[x].Qj←0 }) ; ∀x (if (RS[x].Qk=r) {RS[x].Vk←result ; RS[x].Qk←0 }) ; RS[r].Busy←No ;
Write result of store	Execution complete at r and RS[r].Qk=0	Mem[RS[r].A]←RS[r].Vk ; RS[r].Busy←No ;

The CDB broadcasts its result to many destinations in a single clock cycle.

If the waiting instructions have their operands, they can all begin execution on the next clock cycle.

A Loop Example

The power of Tomasulo's algorithm in handling **WAR** and **WAW** hazards is demonstrated in loops.

```
Loop:   L.D      F0,0(R1)
        MUL.D    F4,F0,F2
        S.D      F4,0(R1)
        DADDIU   R1,R1,-8
        BNE      R1,R2,Loop; branches if R1≠R2
```

If branched are predicted **taken**, RS usage allows multiple executions of the loop to proceed at once.

The loop is unrolled dynamically by HW, using the RSs obtained by renaming to act as additional registers.

No need for compiler unrolling.

Let all the instructions in two successive iterations be issued, but assume that none of the operations within the loop has completed.

Instruction status

Instruction		From iteration	Issue	Execute	Write Result
L.D	F0,0(R1)	1	√	√	
MUL.D	F4,F0,F2	1	√		
S.D	F4,0(R1)	1	√		
L.D	F0,0(R1)	2	√	√	
MUL.D	F4,F0,F2	2	√		
S.D	F4,0(R1)	2	√		

The integer ALU operation is ignored, and it is assumed the branch was predicted as taken.

Reservation station

Name	Busy	Op	Vj	Vk	Qj	Qk	A
Load1	yes	Load			0		Regs[R1] + 0
Load2	yes	Load					Regs[R1] – 8
Add1	no						
Add2	no						
Add3	no						
Mult1	yes	MUL		Regs[F2]	Load1		
Mult2	yes	MUL		Regs[F2]	Load2		
Store1	yes	Store	Regs[R1]		0	Mult1	
Store2	yes	Store	Regs[R1] – 8		0	Mult2	

```
Loop:   L.D     F0,0(R1)
        MUL.D   F4,F0,F2
        S.D     F4,0(R1)
        DADDIU  R1,R1,-8
        BNE     R1,R2,Loop;
```

```
Loop:   L.D      F0,0(R1)
        MUL.D    F4,F0,F2
        S.D      F4,0(R1)
        DADDIU   R1,R1,-8
        BNE      R1,R2,Loop;
```

Register status

Field	F0	F2	F4	F6	...	F30
Qi	Load2		Mult2			

Was Load1 before

Was Mult1 before

Two copies of the loop could be sustained with a CPI close to 1.0, provided MULT completes in 4 clock cycles.

For 6 cycles MULT, more iteration needs be processed before steady state is reached, requiring more FP RSs.

Hazards Through Memory

OOO is safe for access of different addresses.

In same address, if **load precedes store**, interchange results in a **WAR** hazard.

If **store precedes load**, interchange results in a **RAW** hazard.

Interchange of **two stores** results in a **WAW** hazard.

To determine if **load** can be executed, CPU checks (through A fields) whether any preceding **store** (in code order) shares the same memory address.


If conflict is found, load is not sent to the load buffer until the conflicting store completes.

CPU must have computed the A field associated with any earlier memory operation.

Simple solution is to perform the effective address calculations (A field) in code order.

Stores operate similarly, except that CPU checks conflicts in both load and store buffers.

Store must wait until no earlier unexecuted loads or stores sharing the same memory address.

Note that loads can be reordered freely. (why?)

Dynamic scheduling yields very high performance, provided branches are predicted accurately. The major drawback is the HW complexity.

Each **RS** must contain a high speed associative buffer, and complex control logic.

Single **CDB** is a bottleneck. More **CDB**s can be added.

Since each **CDB** must interact with each RS, the associative tag-matching HW must be duplicated at each **RS** for each **CDB**.

Summary: Tomasulo's alg. combines two techniques: renaming of the **ISA** registers to a larger set, and buffering of source operands from the **RF**.


Tomasulo's is widely adopted in **multiple-issue** processors since 1990s.

It achieves high performance without requiring the compiler to target code to a specific pipeline structure.

Cache misses is a major motivations for dynamic scheduling.

OOO execution allows CPU continue executing instructions while awaiting the cache miss completion, hiding some of the miss penalty.

Dynamic scheduling is a key component of **speculation**.